PYE: A Framework for Precise-Yet-Efficient Just-In-Time Analyses for Java Programs

MANAS THAKUR and V. KRISHNA NANDIVADA, IIT Madras

Languages like Java and C# follow a two-step process of compilation: static compilation and just-in-time (JIT) compilation. As the time spent in JIT compilation gets added to the execution-time of the application, JIT compilers typically sacrifice the precision of program analyses for efficiency. The alternative of performing the analysis for the whole program statically ignores the analysis of libraries (available only at runtime), and thereby generates imprecise results. To address these issues, in this article, we propose a two-step (static+JIT) analysis framework called precise-yet-efficient (PYE) that helps generate precise analysis-results at runtime at a very low cost.

PYE achieves the twin objectives of precision and performance during JIT compilation by using a two-pronged approach: (i) It performs expensive analyses during static compilation, while accounting for the unavailability of the runtime libraries by generating partial results, in terms of *conditional values*, for the input application. (ii) During JIT compilation, PYE resolves the conditions associated with these values, using the pre-computed conditional values for the libraries, to generate the final results. We have implemented the static and the runtime components of PYE in the Soot optimization framework and the OpenJDK HotSpot Server Compiler (C2), respectively. We demonstrate the usability of PYE by instantiating it to perform two context-, flow-, and field-sensitive heap-based analyses: (i) points-to analysis for null-dereference-check elimination; and (ii) escape analysis for synchronization elimination. We evaluate these instantiations against their corresponding state-of-the-art implementations in C2 over a wide range of benchmarks. The extensive evaluation results show that our strategy works quite well and fulfills both the promises it makes: enhanced precision while maintaining efficiency during JIT compilation.

CCS Concepts: • **Theory of computation → Program analysis**; • **Software and its engineering → Compilers**; **Just-in-time compilers**; *Dynamic analysis*; Object oriented languages;

Additional Key Words and Phrases: Java program analysis, precise and efficient program analysis

ACM Reference format:
Manas Thakur and V. Krishna Nandivada. 2019. PYE: A Framework for Precise-Yet-Efficient Just-In-Time Analyses for Java Programs. *ACM Trans. Program. Lang. Syst.* 41, 3, Article 16 (July 2019), 37 pages.
https://doi.org/10.1145/3337794

1 INTRODUCTION

Modern languages like Java and C# follow a two-step process for compilation and execution: the input program is statically compiled to an intermediate language (for example, Bytecode for Java and Common Intermediate Language (CIL) for C#), which is then executed on a possibly remote

Authors' addresses: M. Thakur and V. Krishna Nandivada, Department of CSE, IIT Madras; emails: manas@cse.iitm.ac.in, nvk@iitm.ac.in.

0164-0925/2019/07-ART16 $15.00
https://doi.org/10.1145/3337794

```
1  class B {
2    X f;
3    B() {
4      f = new X();
5      f.g = new Y();
6      f.g.h = new Z();
7    }
8    void bar() {
9      B r1 = new B();
10     List r2 = new AList();
11     r2.add(r1);
12     X x = r1.f;
13     Y y = x.g;
14     Z z = y.h;
15     ...
16   }
17 }
```

(a)

```
1  class AList<E> extends List<E> {
2    // AList is a fixed size list.
3    // 1. arr is a final field
4    // allocated in the constructor.
5    // 2. size is a private field
6    // initialized in the constructor.
7
8    void add(E elem) {
9      arr[size++] = elem;
10   }
11 }
```

(b)

Fig. 1. (a) A snippet of a synthetic Java program. (b) Simplified code for the library method `AList.add`. While analyzing the method `bar`, the code for `AList.add` is not available, and vice-versa.

virtual machine (for example, JVM and .NET). Many virtual machines (Alpern et al. 2005; Paleczny et al. 2001) use inbuilt just-in-time (JIT) compiler(s) to generate optimized assembly code that can be directly executed on the hardware. While this can lead to significant performance gains compared to the "interpreter only" mode, it also brings in some interesting challenges.

One of the main challenges in JIT compilation arises from the fact that the time spent in compilation, which includes program-analysis time, gets added to the execution-time of the program. Hence, it is important that the time spent in JIT compilation is not prohibitively high. Consequently, typical JIT compilers in popular virtual machines (such as the HotSpot JVM (Paleczny et al. 2001) and the Jikes RVM (Alpern et al. 2005)) perform imprecise analyses in place of precise whole-program analyses and end up sacrificing precision for efficiency.

An alternative to performing imprecise analyses during JIT compilation is to perform expensive whole-program analyses during static compilation, and use the results during JIT compilation. However, the runtime libraries (such as the JDK) on the machine where the program is executed may differ from those available statically on the machine where the program is compiled. As a result, though this alternative does not impact the JIT compilation time much, the static analyses have to handle calls to library methods in a conservative manner, which may again lead to imprecision.

Thus, both the practical alternatives—(i) whole-program analysis at compile-time and (ii) fast analysis during JIT compilation—may lead to imprecise results. We illustrate these issues in the context of a points-to analysis that is used to remove unnecessary "null-dereference-checks" in Java programs.

In Java, before executing each statement that dereferences an object, the JVM needs to check whether the object being dereferenced is null; if so, a *NullPointerException* must be thrown. Consider the Java code snippet shown in Figure 1. In the method `bar`, the statements 11, 12, 13, and 14 dereference objects. As the variables `r1` and `r2` point-to concrete objects allocated at lines 9 and 10, respectively, the null-dereference checks at lines 11 and 12 can be safely skipped. Further, the null-dereference checks at lines 13 and 14 can also be skipped if (i) `r2` is known to point to an object of type `AList`, and (ii) the method `AList.add` does not modify the object pointed-to via the fields of its parameter. Thus, the number of null-dereference checks that can be skipped (or eliminated) depends directly on the precision of the underlying points-to analysis used. We now discuss the impact of the two above discussed analysis alternatives on null-dereference-check elimination, in the context of the example in Figure 1.

Alternative A1: Analysis during static compilation. A statically performed whole-program flow- and field-sensitive points-to analysis must assume the code of the method `AList.add` as unavailable (else risk the results being unsound). Thus, using the alternative A1, we can elide the null-checks at lines 11 and 12, but not the ones at lines 13 and 14.

Alternative A2: Analysis during JIT compilation. Typical JIT compilers restrict themselves to very imprecise analyses. For example, the points-to analysis used by the HotSpot Server Compiler (C2) is only intraprocedural. Thus, C2 can again elide the null-checks only at lines 11 and 12.

In this article, we propose a two-step analysis framework called PYE ("Precise-Yet-Efficient" framework) that addresses all the issues discussed above. PYE helps generate highly precise analysis-results for application programs during JIT compilation, at a very low cost. We achieve this using a two-pronged approach in PYE: (i) We offload expensive analyses to the static Java compiler, where, in contrast to traditional summaries for each method, we generate "partial summaries." To avoid the imprecision arising out of the unavailable runtime libraries, we propose the novel notion of "conditional values" as a way to store the dependences between the application and the libraries. For example, in the context of null-dereference-check elimination, using traditional simple values, we say that a variable x may point to either a concrete-object or a null-object (concrete- and null-objects are the simple values). In contrast, our proposed conditional values allow us to reach conclusions of the following form: variable x may point to a concrete object if another variable y points to a concrete object, and null otherwise. The partial summaries consist of a set of conditional values for each program element in the method being analyzed. (ii) We pass the output of the static compiler (class files + partial summaries) to the JVM, where the JIT compiler evaluates the conditional values in the partial summaries, after merging the partial summaries of the libraries (pre-computed, once for each library installation), and generates final analysis-results.

PYE addresses the three challenges that can be envisaged in such a multi-step analysis framework: (i) It handles the possible imprecision arising out of the unavailable parts of a program while performing precise whole-program analyses. (ii) It makes sure that the generated partial summaries are succinct and do not lead to any significant storage overhead. (iii) It loads and resolves the partial summaries efficiently without increasing the time spent during JIT compilation.

We have used PYE to design two context-, flow-, and field-sensitive heap-based analyses. The first one is a Points-to Analysis to perform null-Check Elimination (PACE, in short), which elides unnecessary null-dereference checks in Java programs. For the code shown in Figure 1, PACE generates partial summaries, which indicate that while the null-dereference checks at lines 11 and 12 can be unconditionally elided, the same at lines 13 and 14 can be elided only if the method `AList.add` does not assign null to the fields of its first parameter. During JIT compilation, after loading the library-partial-summary (which indicates that the method `AList.add` does not modify any field of the first parameter), PACE resolves the partial summary for `bar` and elides all the null-dereference checks in `bar`. Importantly, PACE achieves high precision without incurring any significant overhead during JIT compilation. We have also used PYE to design an Escape Analysis (Blanchet 2003) and demonstrate its effects on Synchronization Elimination (EASE, in short). Escape analysis finds objects that are local to a thread, and is widely used for eliminating useless synchronization (Blanchet 2003; Choi et al. 1999; Ruf 2000); see Section 2.2 for a brief background on escape analysis. We chose these two analyses because though both are based on pointer analysis, they have different types of lattices, and are quite pedagogical and illustrative of the intricacies involved in their design.

We have implemented the core of the PYE framework as well as the two analyses PACE and EASE in two parts: (i) the components associated with the static compiler—implemented in the Soot optimization framework (Vallée-Rai et al. 1999); and (ii) the components associated with the JIT compiler—implemented in the HotSpot Server Compiler (C2) of the OpenJDK HotSpot JVM (Paleczny et al. 2001).

We have evaluated PYE using PACE and EASE on a series of benchmarks from the SPECjvm (2008), DaCapo (Blackburn et al. 2006), and Java Grande Forum (JGF) (Daly et al. 2001) suites, and SPECjbb (2005). The evaluation shows that the strategy adopted by PYE works quite well: (i) PACE inserts 17.36% fewer null-checks during JIT compilation, on average, than the existing technique employed by C2. (ii) Compared to the existing escape-analyzer of C2 (which elides only 0.03 synchronization operations, on average), EASE elides more synchronization operations (1.13, on average) during JIT compilation. Importantly, compared to the existing analyzers of C2, the improved precision of PACE does not significantly affect the JIT compilation time; in case of EASE, it actually improves the JIT compilation time by 1.9%, on average. Further, the storage overheads for partial summaries are quite low: 6.41% and 3.96% over the class files for PACE and EASE, respectively.

PYE can, in general, be used to perform any whole-program modular dataflow analysis having: (i) a finite-height lattice of dataflow values; (ii) inter-dependent application and library analysis-results; and (iii) dynamically-refinable static-analysis results. Similarly, the discussed points-to and escape analyses can be extended to other respective related JIT optimizations, such as method inlining (Muchnick 1997), garbage collection (Domani et al. 2002), and so on. Though we present PYE in the context of Java, the techniques proposed in this article are general enough to be extended to other languages such as C# that deploy a two-step compilation process.

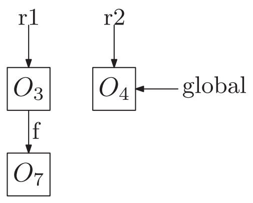
Contributions:

- We propose a new and efficient strategy to obtain precise analysis-results during JIT compilation and formalize it as the PYE framework.
- We introduce the novel notion of *conditional values* as a way to store the dependences between an application and the libraries. These conditional values help us in maintaining *partial summaries* for the application being analyzed statically, and generating final results during JIT compilation without losing precision.
- We instantiate PYE for performing two context-, flow-, and field-sensitive heap-based analyses (PACE and EASE), coupled with optimizations to store the generated partial summaries in a succinct manner and to efficiently process the partial summaries at runtime.
- We demonstrate the efficacy of PYE by performing an extensive evaluation of PACE and EASE in a production Java Virtual Machine (OpenJDK HotSpot JVM) and by comparing the results with those generated by the existing implementations in the JVM. The evaluation shows that PYE fulfills both the promises it makes—enhanced *precision* of analysis-results—while maintaining the *efficiency* of the JIT compiler.

The rest of the article is organized as follows. We first give an overview of some relevant concepts required for further reading in Section 2. We describe our proposed framework PYE along with the novel notion of conditional values in Section 3. We discuss the design of PACE and EASE in Sections 4 and 5, respectively. We highlight some subtle aspects and prove the correctness of PYE in Section 6. We perform a detailed evaluation of PACE and EASE in Section 7. Next, we

```
1  static B global;
2  void foo() {
3      B r1 = new B();
4      B r2 = new B();
5      synchronized(r2) {...}
6      global = r2;
7      r1.f = new B();
8      List lst = new AList();
9      lst.add(r1);
10     B x = r1.f;
11     synchronized(x) {...}
12 }
```

(a)



(b)

Fig. 2. (a) The method `foo` of class `B` shown in Figure 1. (b) The corresponding points-to graph after line 7.

discuss various related works in Section 8. Finally, we conclude the article and highlight some future directions in Section 9.

2 BACKGROUND

In this section, we give a brief description of points-to analysis, thread-escape analysis, and some related data structures that we will be using throughout the article.

2.1 Points-to Analysis

Points-to analysis is a program-analysis technique that establishes which pointers, or reference variables, can point to which objects, or storage locations. The results obtained by a points-to analysis are key to several other heap analyses and related optimizations; for example, alias analysis, shape analysis, call-graph construction, method inlining, and so on.

We represent objects with the line number at which they are allocated. We say a variable *var* may point-to a set S, if the elements of the set S represent the objects that may be pointed-to by the variable during program execution. For example, in the code shown in Figure 2(a), the may-points-to sets of the reference variables `r1` and `r2` are $\{O_3\}$ and $\{O_4\}$, respectively.

2.2 Thread-escape Analysis

Thread-escape analysis (Blanchet 2003), hereafter called escape analysis, partitions the objects allocated in a thread t into two categories: (i) those that are local to t (that is, do-not-escape); and (ii) those that can be accessed by threads other than t (that is, escape). An object may escape to other threads if it is reachable (possibly via a sequence of field dereferences) from a static (global) variable, or from a thread object. Escape analysis has many applications: synchronization elimination (Blanchet 2003; Choi et al. 1999; Ruf 2000), data-race detection (Choi et al. 2002), efficient garbage-collection (Domani et al. 2002), and so on. For example, the synchronization operation in the Java synchronization statement "`L: synchronized(`v`) S`" can be elided if the escape analysis finds that the object(s) pointed to by v does not escape before `L`.

Consider the Java code snippet shown in Figure 2(a). Assume that the code shown by "..." does not affect the heap. In Figure 2(a), the objects O_3, O_7, and O_8 do not escape (as the method `AList.add` in Figure 1 does not make the objects reachable from its parameters escape). Further, O_4 does not escape till line 6. Thus, the synchronization operations at lines 5 and 11 can be safely elided.

2.3 Points-to Graphs

Points-to graphs and their variations are widely used (Dietrich et al. 2015; Sălcianu and Rinard 2005; Tan et al. 2017; Whaley and Rinard 1999) for representing the points-to relations in Java programs. A points-to graph $G(N, E)$ comprises of (i) a set N of nodes that represent variables and abstract objects in the program; and (ii) a set E of edges that represent points-to relationships among the nodes in the program. An edge can optionally have a label representing the field in the corresponding points-to relationship. An edge (a, O_x) from a reference variable a to a node O_x in a points-to graph implies that the variable a may point to the object O_x. Similarly, an edge $(O_x, \texttt{f}, O_y)$ from node O_x to O_y with a label `f` implies that $O_x.\texttt{f}$ may point to O_y.

In this article, while analyzing a method m, we use the points-to graph G_m as a map that returns the current points-to information as follows: (i) $G_m(a)$ returns the points-to set of the variable a; and (ii) $G_m(O_x, \texttt{f})$ returns the points-to set of $O_x.\texttt{f}$. Figure 2(b) shows the points-to graph after line 7 for the code shown in Figure 2(a). The points-to sets represented by the graph are: $G_{\texttt{foo}}(\texttt{r1}) = O_3$, $G_{\texttt{foo}}(\texttt{r2}) = O_4$, $G_{\texttt{foo}}(\texttt{global}) = O_4$, and $G_{\texttt{foo}}(O_3, \texttt{f}) = O_7$.

Points-to graphs can also be used to perform escape analysis by checking whether a node is reachable from static variables or nodes representing thread objects. Given a points-to graph G_m, we use a function $G_m.\mathit{reachables}(a)$ to get the nodes reachable from a in G_m. In Figure 2(b), $O_4 \in G_{\texttt{foo}}.\mathit{reachables}(\texttt{global})$, and hence, O_4 in method `foo` escapes its allocating thread.

3 THE PYE FRAMEWORK

In this section, we first briefly discuss some of the challenges in typical modular dataflow-analysis techniques, and then we describe PYE in the context of analyzing Java applications.

3.1 Typical Modular Analyses

To maintain scalability, typical modular analyses (Choi et al. 1999; Whaley and Rinard 1999) process one method at a time and maintain its *summary*. For a given dataflow analysis Ψ, the summary of a method m can be seen as a map f_m from the domain $\mathcal{D}$ of Ψ to the set of dataflow values `Val` of Ψ. That is:

$$f_m : \mathcal{D} \rightarrow \texttt{Val} \tag{1}$$

We assume that `Val` forms a lattice with a meet operation $\sqcap$, a supremum $\top$ (the most precise element), and an infimum $\bot$ (the most conservative element). For example, in typical escape analysis algorithms (Blanchet 2003; Bogda and Hölzle 1999; Ruf 2000), (i) $\mathcal{D}$ includes object-allocation sites, function parameters, and return values; and (ii) the lattice `Val` has elements from the set $\{\top, \bot\}$, organized as a chain, indicating the escape-status ($\top$ = *DoesNotEscape*, and $\bot$ = *Escapes*).

For a method m, its summary f_m may depend on the summaries of a set of other methods. Thus, to compute f_m precisely, all the dependent summaries must be available. In the context of JIT compilation (for example, in the HotSpot JVM (Paleczny et al. 2001)), the summaries dependent on the runtime-libraries can only be computed at runtime. This can usually be achieved using one of the two approaches shown in Figure 3(a). A JIT compiler can perform either very precise analyses and incur the large overheads caused by the compilation time; or it can target fast compilation time and perform imprecise analyses. Note that there could be several other configurations that explore the tradeoffs between these two approaches such as k-limited context-sensitivity (Sharir and Pnueli 1981), flow-insensitivity (Hardekopf and Lin 2007), cutoff-based approaches (Vivien and Rinard 2001), and so on. However, for simplicity and efficiency, typical JIT compilers (such as the ones in the HotSpot (Paleczny et al. 2001) and Jikes (Alpern et al. 2005) virtual machines) limit themselves to mostly intraprocedural analyses. Even though many JIT compilers may perform early-inlining,

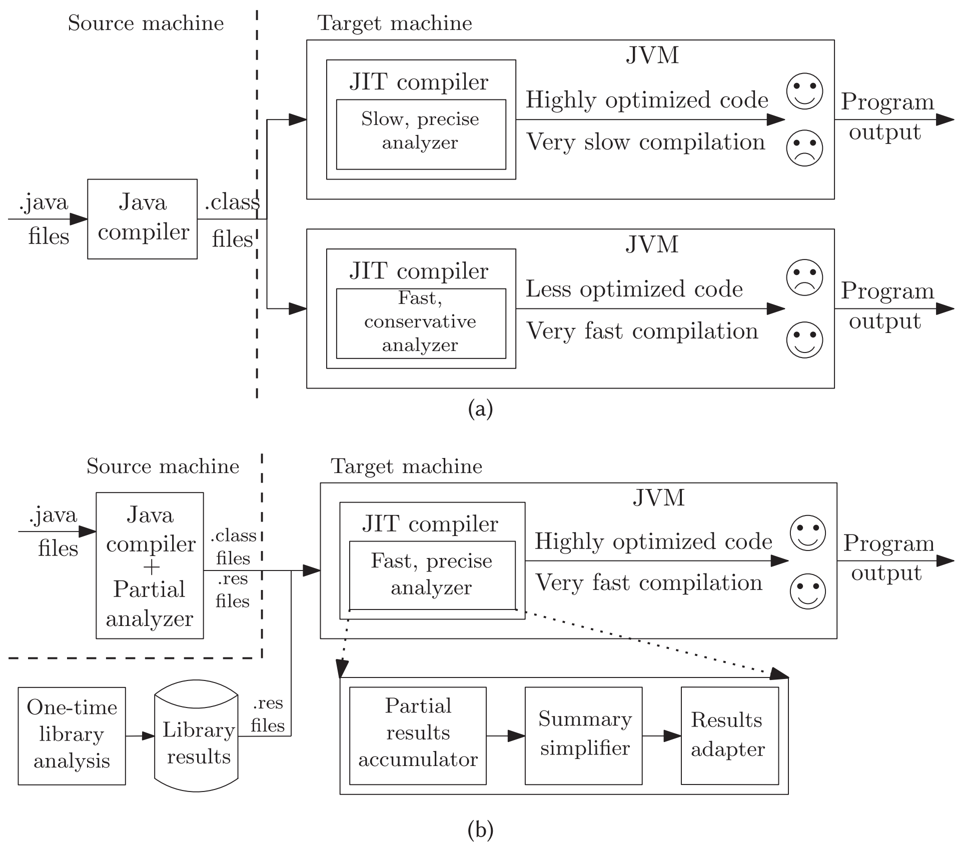


Fig. 3. Java program analysis: (a) Two traditional approaches. (b) Approach adopted by PYE.

its impact on the precision of the analysis is limited due to the standard restrictions (Paleczny et al. 2001) on inlining (such as the iCache size, deep nesting of methods, recursion, profiling, and so on).

An alternative to costly analyses during JIT compilation is to perform the analysis statically at compile time and export the results to the JVM. However, as the JDK installation on the source machine (where the analysis is performed) may be different from the target machine (on which the analysis-results will be used), using such results may lead to unsound optimizations. Examples of such changes include removal/addition of methods (for example, compared to Java 8, *LogManager.addPropertyChangeListener* was removed in Java 9), changes in method signatures, newer implementations overriding parent-class methods, and so on. As a result, all the referred library methods are considered unavailable at compile time, and the summary of each library method is conservatively approximated to the special "bottom" function $\lambda x.\bot$. Such a scheme can lead to overly imprecise results.

3.2 PYE: A Practical Alternative

To overcome the issues of both (i) fully static analysis (too imprecise), and (ii) whole-program analysis at runtime (prohibitively expensive), we propose PYE: a framework for **p**recise-**y**et-**e**fficient just-in-time analyses for Java programs. Figure 3(b) shows the block diagram of PYE. Compared to the traditional scheme of analysis in the JIT compiler, PYE analyzes an application using two inter-related components: the *partial-analyzer* (added to the static Java compiler), and the *fast-precise-analyzer* (added to the JIT compiler of the JVM).

For each method in the application being analyzed, the partial-analyzer skips the calls to the unavailable library methods and generates *partial summaries*. Compared to a traditional method-summary f_m (a map from the domain $\mathcal{D}$ to one of the dataflow values in `Val`), a partial summary maps each element in the domain of the analysis to a set of *conditional values*. We propose the novel notion of conditional values as a way to encode the dependence of the analysis-results for the associated program element on other unavailable program element(s). For each application, the partial summaries generated by the partial-analyzer are stored as a ".res file". Similarly, another instance of the partial-analyzer analyzes the library methods of the target machine offline, independent of the application, and generates a partial summary for each library method. As the static-compilation time does not get added to the execution-time of the program, an analysis-writer using PYE is free to pick highly precise variants of analyses to be performed by the partial-analyzer.

As shown in Figure 3(b), when a program is executed, along with the .class files (of the application and the libraries), the corresponding partial summaries in the form of .res files are made available to the JVM on the target machine. In the JIT compiler, the fast-precise-analyzer reads the required partial summaries (*partial-results-accumulator*), resolves the dependences between the application and the library summaries to generate final results (*summary-simplifier*), and then populates the appropriate JVM data structures to perform the related optimizations (*results-adapter*).

We first describe the notion of conditional values as a way to encode the dependences between various parts of a program, and define partial summaries. Later, we describe how these partial summaries are simplified to obtain precise results during JIT compilation.

3.3 Partial Summaries

For a given dataflow analysis Ψ, to compute the summary f_m for each method m, the partial-analyzer of PYE first computes the *partial summary* g_m, which is a map from the domain $\mathcal{D}$ of Ψ to the power set of *conditional values* (`CVal`):

$$g_m : \mathcal{D} \rightarrow P(\texttt{CVal}) \tag{2}$$

Definition 1. A conditional value is a three-tuple of the form

$$\mathcal{T} = \langle \Theta, val, val' \rangle,$$

where:

- Θ is a pair of the form $\langle u, x \rangle$, in which:
  - u is a method
  - x is a program element in u
- val and val' are elements from the set `Val` of Ψ

A conditional value $\mathcal{T} = \langle \langle u, x \rangle, val, val' \rangle$ in $g_m(e)$ encodes a condition on the final analysis-result $f_u(x)$ for the element x in method u. That is, if $f_u(x) = val$, then $\mathcal{T}$ *evaluates* to val'. If u is a library method, $f_u(x)$ is not available statically while analyzing the application; hence, it is not possible to resolve the dependence till runtime.

Example. Consider the code snippet in Figure 4. Suppose the goal is to perform a flow-analysis that maps each reference variable (such as `p` and `q` in the example) to the set of classes whose objects could be pointed-to by the variable. The parameter `p` could either point to an object of class `D1` or of class `D2`. Considering the flow-analysis starting from the method `foo`, the set of conditional values generated by the partial-analyzer for `q` would be:

$$g_{\texttt{foo}}(\texttt{q}) = \{\langle\langle \texttt{foo}, \texttt{p}\rangle, \{\texttt{D1}\}, \{\texttt{A1}\}\rangle, \langle\langle \texttt{foo}, \texttt{p}\rangle, \{\texttt{D2}\}, \{\texttt{A2}\}\rangle\}$$

```java
class C {
   void foo(D p) {
      A q = p.bar();
   }
}
```

```java
interface D { A bar(); }
class D1 implements D {
   A bar() {
      return new A1();
   }
}
class D2 implements D {
   A bar() {
      return new A2();
   }
}
```

Fig. 4. A synthetic Java code snippet.

Here, the conditional values indicate that the flow-set of q would include A1 if the flow-set of p is {D1}, and the flow-set of q would include A2 if the flow-set of p is {D2}. In Section 3.4, we explain how such conditional values are simplified in a systematic manner.

Notation. If the analysis-result of an element e does not depend on any other element, then $g_m(e)$ is a singleton with a lone conditional value whose condition is a tautology. We denote the set of such *simple* ("vacuously true") conditional values as SCVal, which can be seen as the conditional representation of the set Val. We represent each $v \in$ Val as a conditional value $\langle \Theta_v, v, v \rangle \in$ SCVal, where Θ_v can be seen as a special global element for which $f_m(\Theta_v)$ is set to v for all methods m. We refer to the rest of the conditional values, that is, the ones in the set CVal−SCVal, as *dependent* conditional values, DCVal.

Say x and y are program elements in methods u and m, respectively. In analyses like PACE (Section 4) and EASE (Section 5), in each conditional value, the respective third and the fourth components match, and if $\exists\ v \in$ Val, such that $\langle\langle u, x\rangle, v, v\rangle \in g_m(y) \Rightarrow \forall v' \in$ Val, $\langle\langle u, x\rangle, v', v'\rangle \in g_m(y)$. For such analyses, for brevity, we abbreviate the set of all the conditional values dependent on $\langle u, x\rangle$, by only $\langle u, x\rangle$. For uniformity, a simple conditional value $\langle \Theta_v, v, v\rangle \in$ SCVal in those analyses is abbreviated to Θ_v.

Example. We now illustrate the above discussed concepts using another analysis that performs a points-to analysis to elide the null-dereference-checks for which the dereference is guaranteed to be performed on a concrete object. Say the set Val for this analysis is {C (for *Concrete*), N (for *Null*)}. The simple conditional values corresponding to C and N are Θ_C and Θ_N, respectively. We use the code shown in Figure 1 as the input for this analysis. The conditional values generated by the partial-analyzer for each dereference O_l (denoting the dereference at line l) in the method bar are:

$$g_{\texttt{bar}}(O_{11}) = g_{\texttt{bar}}(O_{12}) = \{\Theta_C\}$$
$$g_{\texttt{bar}}(O_{13}) = \{\langle \texttt{AList.add}, O_{p_1}.\texttt{f}\rangle\}$$
$$g_{\texttt{bar}}(O_{14}) = \{\langle \texttt{AList.add}, O_{p_1}.\texttt{f.g}\rangle\}$$

Here, as the variables r1 and r2 point to concrete objects allocated at lines 9 and 10, respectively, the set of conditional values for the dereferences at lines 11 and 12 is the singleton Θ_C. The conditional value for the dereference at line 13 indicates that the dereferenced object would be null if the object pointed-to by O_{p_1}.f, where O_{p_1} is the object pointed-to by the first parameter of AList.add, is null, and concrete otherwise. Similarly, the conditional value for the dereference at line 14 indicates that the dereferenced object would be null (or concrete) if the object pointed-to by O_{p_1}.f.g is null (or concrete).

3.4 Simplifying Partial Summaries

The partial-analyzer generates a partial summary g_m for each method m statically available for analysis. Say the set of all such summaries for an application $\mathcal{A}$ is $\mathcal{F}_{\mathcal{A}}$. On the target machine, another instance of the partial-analyzer computes a similar summary for each library method offline; say the combined set for a library installation $\mathcal{L}$ is $\mathcal{F}_{\mathcal{L}}$. At runtime, the summaries generated by the partial-analyzer for all the methods (application as well as library) are available. When the application is executed by the JVM on the target machine, the fast-precise-analyzer of PYE takes $\mathcal{F}_{\mathcal{A}}$ and $\mathcal{F}_{\mathcal{L}}$, and computes the final analysis-results (in the summary-simplifier; see Figure 3(b)) for all the elements of a method m that is compiled just-in-time by the JVM.

For each element e in the analysis-domain for the method m being compiled by the JIT compiler, the summary-simplifier of the fast-precise-analyzer evaluates each conditional value $\mathcal{T} = \langle\langle u, x\rangle, val, val'\rangle \in g_m(e)$ by looking up the value of $f_u(x)$ and returns the evaluated value $[\![\mathcal{T}]\!] \in$ `Val`. If the condition specified in $\mathcal{T}$ evaluates to true (that is, $f_u(x) = val$), then $[\![\mathcal{T}]\!]$ is val'; else, $[\![\mathcal{T}]\!]$ is $\top$ (the top value of the lattice `Val`):

$$[\![\langle\langle u, x\rangle, val, val'\rangle]\!] = (f_u(x) == val)\,?\,val' \;:\; \top \tag{3}$$

For analyses where val is always equal to val', we use the shorthand notation introduced in Section 3.3, and simplify Equation (3) as:

$$[\![\langle u, x\rangle]\!] = f_u(x) \tag{4}$$

Finally, given the set of evaluated conditional values $S = \{[\![\mathcal{T}]\!] \mid \mathcal{T} \in g_m(e)\}$, the analysis-result $f_m(e)$ is computed as the meet over all the elements in S. The semantics of the meet operation is specific to the individual analysis under consideration. Thus, if the meet operation of an analysis is denoted by $\sqcap$, then:

$$f_m(e) = \sqcap_{\forall \mathcal{T} \in g_{m(e)}} [\![\mathcal{T}]\!] \tag{5}$$

Given a set of program elements and their conditional values, evaluating all the conditional values may require repeated solving of Equations (3) (or (4)) and (5) till a fixed-point. Such a fixed-point computation is necessary to take into consideration the dependence of the conditional values of one program element on those of other program elements. We do so by using a standard worklist-based algorithm in the summary-simplifier of the fast-precise-analyzer. As `Val` is a finite-height lattice, the evaluation is guaranteed to terminate and give us the most precise solution for the conditional values generated by the partial-analyzer. Presence of un-evaluated conditional values even after attaining a fixed-point indicates mutually cyclic dependences. In such cases, we use $f_m(e) = \top$ for each element involved in the cycle (as we have already achieved a fixed-point, it is safe to do so). The cost of performing this fixed-point computation mostly depends on the number of dependent conditional values (usually a small percentage compared to the total number of program elements) generated for a particular analysis. Further, the amortized cost required to resolve a dependence (one lookup per dependence) is $O(1)$. As we show in Section 7, for the analyses under consideration, the time spent in summary-simplification is very small (order of milliseconds).

Example. For the code shown in Figure 1, the partial summary generated by the partial-analyzer for the method `AList.add`, after analyzing the library offline, would be:

$$g_{\texttt{AList.add}}(O_9) = \{\Theta_C\},$$

which indicates that the dereference performed at line 9 in `AList.add` is done on a concrete object, and the absence of any information about its parameters implies that none of them are modified in the method. Note: The meet ($\sqcap$) operation for this analysis is defined as: $\sqcap(C, C) = C$ and $\sqcap(C, N) = \sqcap(N, C) = \sqcap(N, N) = N$. While compiling the method `bar`, the fast-precise-analyzer looks up the partial summary of the method `AList.add` to resolve the conditions for the elements of `bar`, and it generates the following final analysis-results for the dereferences in `bar` (after solving Equations (4) and (5)):

$$f_{\texttt{bar}}(O_{11}) = f_{\texttt{bar}}(O_{12}) = \sqcap\{[\![\Theta_C]\!]\} = \sqcap\{C\} = C$$
$$f_{\texttt{bar}}(O_{13}) = \sqcap\{[\![\langle \texttt{AList.add}, O_{p_1}.\texttt{f}\rangle]\!]\} = \sqcap\{C\} = C$$
$$f_{\texttt{bar}}(O_{14}) = \sqcap\{[\![\langle \texttt{AList.add}, O_{p_1}.\texttt{f.g}\rangle]\!]\} = \sqcap\{C\} = C$$

// $\langle \texttt{AList.add}, O_{p_1}.\texttt{f}\rangle$ and $\langle \texttt{AList.add}, O_{p_1}.\texttt{f.g}\rangle$ cannot be simplified further,
// and hence evaluate to $\top = C$.

The results indicate that all the dereferences in the method `bar` are done on concrete objects, and, hence, the corresponding null-pointer-dereference checks can be safely elided. Compared to the results generated by the analysis alternatives A1 and A2 in Section 1, it is evident that PYE is able to achieve a higher precision by combining the partial summaries for the application and the library at runtime. We show in Section 7 that this precision comes at a very low cost; that is, the associated overheads are very small.

3.5 Efficient Storage of Partial Summaries

The partial summaries generated by the partial-analyzer of PYE for each application are stored in a .res file on the machine where the analysis is performed. This .res file needs to be transferred to the target machine, along with the .class files for the application. On the target machine, the .res files for the application and the libraries are read by the JVM during execution. The speed of performing all the above operations depends a lot on the size of these .res files. Thus, the smaller the .res files, the fewer will be the storage, transfer, and file-reading overheads. For each analysis implemented in PYE, in order to efficiently maintain and store the partial summaries, we perform an optimization to pre-apply the meet operation in the partial-analyzer: For each method m, for each $g_m(x)$, we pre-apply the meet operation on the conditional values and store either only a single simple conditional value, or one or more dependent conditional values along with at most one simple conditional value. For example, consider the null-check removal analysis discussed in Section 3.3. Say, for an object O in the method m, a dependent conditional value $\langle u, x\rangle$ gets added to $g_m(O)$. If $g_m(O)$ previously consisted of a simple conditional value Θ_C, we remove Θ_C from $g_m(O)$ and only keep $\langle u, x\rangle$. This optimization reduces the number of conditional values we carry while performing the static analysis.

3.6 Writing an Analysis in PYE

PYE can, in general, be used to perform any whole-program modular dataflow analysis (for languages like Java/C#) having: (i) a finite-height lattice of dataflow values, (ii) inter-dependent application and library analysis-results, and (iii) dynamically-refinable static-analysis results. Examples include the inclusion-based points-to analysis (Andersen 1994), unification-based analysis (Steensgaard 1996), partial escape analysis (Stadler et al. 2014), MHP analysis (Naumovich et al. 1999), and so on. In this section, we give an overview of how to implement an existing analysis Ψ in PYE. First, the analysis writer needs to specify the domain $\mathcal{D}$ of Ψ, and the lattice formed by the dataflow values `Val` of Ψ. In the modified analysis, each value $v \in$ `Val` is converted to a special conditional value of the form Θ_v. As part of the partial-analyzer, the modified analysis then processes each statement similar to Ψ, except for the following three scenarios, which need to take into consideration the generation of conditional values:

(i) *Unavailable callee.* Say, while analyzing a method, we encounter a call to an unavailable method (say from a library). Here, the analysis writer needs to encode the dependence of the actual arguments on the method u, using conditional values.

(ii) *Unavailable caller.* Say we start analyzing a method in an unknown calling-context, for example, starting the analysis of a library method u. Here, the analysis writer needs to encode the dependence of the formal parameters on the actual arguments that may be passed to the method u, using conditional values.

(iii) *Unavailable object-dereference.* Say we encounter a load statement of the form $a = b.f$, and b depends on another element from an unavailable method, for example, b holds the return value of a library method u. Here, the analysis writer needs to encode the dependence of $b.f$ on the method u using conditional values.

As an example of how the dependences need to be encoded, consider the call to the unavailable method `AList.add` at line 9 in Figure 2(a). For the object O_3 pointed-to by `r1`, at this call, a traditional static escape analysis would record the escape-status of O_3 to be the value E (or *Escapes*). Whereas the same analysis implemented in PYE would record the fact that the escape-status of O_3 depends on the first parameter of `AList.add`, using the conditional value $\langle \texttt{AList.add}, O_{p_1} \rangle$.

In addition to the above changes in the partial-analyzer, the analysis writer needs to provide the implementation of the results-adapter in the fast-precise-analyzer. This simply involves populating the appropriate data structures in the JVM from the results generated by the summary-simplifier, such that they can be accessed directly by the optimizers of the JIT compiler. Note that an otherwise fully just-in-time analysis, in addition to writing the analysis, also needs to appropriately populate the data structures for any dependent optimization passes, and, hence, the effort required to do the same in PYE is arguably never more.

Overall, PYE achieves the precision of a whole-program analysis with very low analysis overheads at runtime. As it can be seen, PYE replaces complex program-analysis phases of the JIT compiler with basic operations like reading the pre-computed summaries and simplifying the summaries based on the summaries of other methods. This strategy pays off quite well by improving the precision of analysis-results without significantly affecting the time required for JIT compilation (see Section 7).

4 POINTS-TO ANALYSIS FOR NULL-CHECK ELIMINATION IN PYE

In this section, we illustrate the usage and effectiveness of PYEby using it to efficiently perform a top-down context-, flow-, and field-sensitive points-to analysis for null-pointer-dereference check elimination (or PACE, in short) in Java programs. The analysis is based on points-to graphs (see Section 2 for an overview) and is used to remove the implicit null-dereference checks in translated Java programs for the dereferences that are guaranteed to be made on concrete objects (as discussed in Section 1).

As mentioned in Section 3.6, in order to perform an analysis using PYE, we need to specify the set $\mathcal{D}$, the lattice formed by `Val`, the processing of each relevant statement by the partial-analyzer, and the results-adapter. We now describe the same for PACE.

4.1 Partial-analyzer for PACE

Figure 5 shows the domain $\mathcal{D}_{pace}$ of relevant program elements and the lattice of dataflow values $\texttt{Val}_{pace}$ for PACE. For each method m, $\mathcal{D}_{pace}$ consists of six sets of abstract objects: (i) $\texttt{ALC}_m$: one object O_l per allocation statement labeled l; (ii) $\texttt{PAR}_m$: one object O_{p_i} representing the objects pointed-to by the parameter p_i; (iii) $\texttt{RET}_m$: all the objects returned by m; (iv) $\texttt{OUT}_m$: one object $O_{d@l}$ for an unavailable dereference at a statement labeled l; (v) $\texttt{UCS}_m$: one object $O_{u@l}$

Set		Description
$\texttt{ALC}_m$	=	$\{O_l \mid l$ is an allocation statement in method $m\}$
$\texttt{PAR}_m$	=	$\{O_{p_i} \mid p_i$ is the i^{th} parameter of method $m\}$
$\texttt{RET}_m$	=	$\{O_x \mid O_x$ is returned by $m\}$
$\texttt{OUT}_m$	=	$\{O_{d@l} \mid O_{d@l}$ represents an unavailable object dereference at line l in $m\}$
$\texttt{UCS}_m$	=	$\{O_{u@l} \mid l$ is a call-statement in method m and the callee u is unavailable$\}$
$\texttt{DRF}_m$	=	$\{O_l \mid l$ is an object-dereferencing statement in method $m\}$ *// specific to PACE*
$\texttt{SYN}_m$	=	$\{O_l \mid l$ is a synchronization statement in method $m\}$ *// specific to EASE*
		Domain of program elements
$\mathcal{D}_{pace}$	=	$\forall m\ \texttt{ALC}_m \cup \texttt{PAR}_m \cup \texttt{RET}_m \cup \texttt{OUT}_m \cup \texttt{UCS}_m \cup \texttt{DRF}_m$
$\mathcal{D}_{ease}$	=	$\forall m\ \texttt{ALC}_m \cup \texttt{PAR}_m \cup \texttt{RET}_m \cup \texttt{OUT}_m \cup \texttt{UCS}_m \cup \texttt{SYN}_m$
		Lattice of dataflow values
$\texttt{Val}_{pace}$	=	$\{Concrete\ (C\ or\ \top),\ Null\ (N\ or\ \bot)\}$
$\texttt{Val}_{ease}$	=	$\{DoesNotEscape\ (D\ or\ \top),\ Escapes\ (E\ or\ \bot)\}$
Meet	:	$\top \sqcap \top = \top;\ \top \sqcap \bot = \bot \sqcap \top = \bot \sqcap \bot = \bot$

Fig. 5. The definitions of $\mathcal{D}$ and `Val` for the analyses PACE and EASE.

per unavailable method u at a call-statement labeled l, indicating the return-value of u at l; and (vi) $\texttt{DRF}_m$: one object O_l per object-dereferencing statement labeled l.

The set $\texttt{Val}_{pace}$ forms a lattice with two elements: *Concrete* (C or $\top$) and *Null* (N or $\bot$). The corresponding conditional values are Θ_C and Θ_N, respectively. The definition of the meet ($\sqcap$) operation is standard (see Figure 5).

Our static analysis is a standard top-down, forward, context-, flow-, and field-sensitive iterative dataflow analysis. The analysis of an application begins at the entry of the main method of the application. For analyzing the libraries (on the target machine), we start the analysis afresh at the entry of each public method of the library. For simplicity, we assume that each intraprocedural Java statement is in a "three address" representation, and that a field-dereference occurs to the right-hand side of an assignment only in a load statement of the form $a = b.f.$ Further, we skip a detailed discussion of statements involving array references and briefly highlight the changes required to process them, if any, while discussing the statements of a similar form. For the ease of analysis, we assume that each method has two special statements `entry` and `exit`, denoting the single point of entry and exit, respectively. We also assume that each statement has a unique label associated with it.

For each method m, we maintain two data structures before and after each statement: (i) a points-to graph G_m (see Section 2 for an overview); and (ii) the partial summary g_m, which is a map from abstract objects to a set of conditional values. We use a worklist-based algorithm and analyze the statements of a method repeatedly till a fixed-point. After analyzing a method m, instead of storing the analysis information at each program point, we store the points-to graph (standard rules) and the partial summary as observed at the `exit` of m. Even then, we realize flow-sensitivity as we separately track each of the object dereferences (the set $\texttt{DRF}_m$). While this increases the number of objects in the points-to graph by $O(N)$, where N is the program size, overall this scheme reduces the amount of stored information, as we avoid storing the points-to graph at each instruction.

We now describe the processing of each statement that could affect either the points-to graph or the set of conditional values for any element. Figure 6 shows the inference rules for updating the points-to graph G_m and the partial summary g_m while analyzing the statements of a method m.

[allocation]	$l : a = new\ B();$	$\begin{cases} g_m[O_l \leftarrow \{\Theta_C\}] \\ G_m[a \leftarrow \{O_l\}] \end{cases}$
[null-assignment]	$a = null;$	$\begin{cases} G_m[a \leftarrow \{O_{null}\}] \end{cases}$
[copy]	$a = b;$	$\begin{cases} G_m[a \leftarrow G_m(b)] \end{cases}$
[store]	$l : a.f = b;$	$\begin{cases} \forall O_a \in G_m(a) \\ \quad G_m[(O_a, f) \cup\!\leftarrow G_m(b)] \\ \texttt{updateDeref}(O_l, a) \end{cases}$
[load]	$l : a = b.f;$	$\begin{cases} G_m[a \leftarrow \cup_{\forall O_b \in G_m(b)} G_m(O_b, f)] \\ \texttt{updateDeref}(O_l, b) \\ \text{if } (\exists O_b \in G_m(b),\ s.t.\ G_m(O_b, f) = \emptyset) \text{ then} \\ \quad \forall \langle u, x\rangle \in g_m(O_b) \cap \texttt{DCVal} \\ \qquad g_m[O_{d@l} \cup\!\leftarrow \langle u, x.f\rangle] \\ \qquad G_m[(O_b, f) \cup\!\leftarrow \{O_{d@l}\}] \\ \qquad G_m[a \cup\!\leftarrow O_{d@l}] \end{cases}$
[throw]	$l : throw\ a;$	$\begin{cases} \texttt{updateDeref}(O_l, a) \end{cases}$
[synch]	$l : synchronized(a)$	$\begin{cases} \texttt{updateDeref}(O_l, a) \end{cases}$
[array-length]	$l : k = a.length;$	$\begin{cases} \texttt{updateDeref}(O_l, a) \end{cases}$
[return]	$return\ a;$	$\begin{cases} \texttt{RET}_m \leftarrow \texttt{RET}_m \cup G_m(a) \end{cases}$
[unavailable-callee]	$l : b = a_0.u(a_1, ..., a_n);$	$\begin{cases} \forall O_{a_i} \in G_m(a_i), \forall O_x \in G_m(O_{a_i}, f) \\ \quad \texttt{extendCVals}(O_x, \langle u, O_{p_i}.f\rangle) \\ g_m[O_{u@l} \leftarrow \{\langle u, O_u\rangle\}] \\ G_m[b \leftarrow \{O_{u@l}\}] \\ \texttt{updateDeref}(O_l, a_0) \end{cases}$
[unavailable-caller]	$m(B\ p_1, ..., B\ p_n)$	$\begin{cases} g_m[O_{p_i} \leftarrow \{\langle \overline{m}, O_{a_i}\rangle]\} \\ G_m[p_i \leftarrow \{O_{p_i}\}] \end{cases}$

Fig. 6. Partial-analysis rules for PACE. Notation: (i) $\beta[O_x \leftarrow Y]$ means β is extended with $\beta(O_x)$ set to Y. (ii) $\beta[O_x \cup \leftarrow Y]$ is an abbreviation for $\beta[O_x \leftarrow \beta(O_x) \cup Y]$.

—*Allocation,* $l : a = new\, B()$. We use the abstract object $O_l \in \texttt{ALC}_m$ to represent the object(s) allocated at line l; the conditional value associated with O_l is set to Θ_C, denoting that O_l is a concrete object. We then set the points-to set of a to $\{O_l\}$.

—*Null-assignment,* $a = null$. In case of an explicit assignment of *null* to a variable a, we set the points-to set of a to a set containing the special object O_{null} (for which $g_m(O_{null})$ is set to $\{\Theta_N\}$), implying that a points to null.

—*Copy,* $a = b$. Here, we set the points-to set of a to that of b.

—*Store,* $l : a.f = b$. First, for each object O_a in $G_m(a)$, we add the objects in the points-to set of b to the points-to set of $O_a.f$. Next, to denote the dereference done at l, we call a macro $\texttt{updateDeref}(O_l, a)$ (see Figure 7(a)), where $O_l \in \texttt{DRF}_m$ represents the object(s) being dereferenced at l. For each object O_a in the points-to set of a, the macro $\texttt{updateDeref}(O_l, a)$ adds all the conditional values in $g_m(O_a)$ to $g_m(O_l)$.

—*Load,* $l : a = b.f$. Here, for each object O_b pointed-to by b, we first add the objects in the points-to set of $O_b.f$ to the points-to set of a, and then handle the dereference done at l by calling the macro $\texttt{updateDeref}(O_l, b)$. In case there exists an abstract object O_b in $G_m(b)$ such that the set $G_m(O_b.f)$ is empty (for example, when O_b is an object returned by an unavailable method), then each dependent conditional value $\langle u, x\rangle$ in $g_m(O_b)$ indicates that O_b depends on the element x of

```
Macro updateDeref($O_l$, $a$)
    $g_m(O_l) \leftarrow \emptyset$;
    foreach $O_a \in G_m(a)$ do
        $g_m(O_l) \cup\!\!\leftarrow g_m(O_a)$;
```

(a)

```
Macro extendCVals($O_x$, $\langle u, e\rangle$)
    $g_m(O_x) \cup\!\!\leftarrow \{\langle u, e\rangle\}$;
    foreach edge $(O_x, f, O_y)$ in $G_m$ do
        if $\neg O_y.visited$ then
            $O_y.visited \leftarrow true$;
            extendCVals($O_y$, $\langle u, e.f\rangle$);
```

(b)

Fig. 7. Macros used in Section 4. (a) `updateDeref`. (b) `extendCVals`. G_m is the current points-to graph and g_m is the current partial summary.

an unavailable method u. In such a case, $O_b.f$ might be modified in u. To denote this dependence, we add the abstract object $O_{d@l} \in \texttt{OUT}_m$ to $G_m(O_b, f)$, and add $\langle u, x.f\rangle$ to $g_m(O_{d@l})$. Finally, we add $O_{d@l}$ to the points-to set of a.

If a store or load is made to/from an array (that is, $a[i] = b$ or $a = b[i]$, respectively), we repeat the same steps as done for a normal store or load statement, except that instead of f, we use the special array field "[]". See Section 6 for a discussion on our choice of heap abstraction.

—*Other dereference statements*. For Java statements that involve an implicit null-dereference, such as l : *throw* a, $l : k = a.length$ and l : *synchronized*$(a)\{\ldots\}$, we use the object $O_l \in \texttt{DRF}_m$ to denote the dereference and call the macro `updateDeref`(O_l, a) to update $g_m(O_l)$.

—*Return, return a*. For each method m, we maintain a set $\texttt{RET}_m$ containing the objects that could be returned by m. At a return statement *return a* in method m, we add all the objects in $G_m(a)$ to the set $\texttt{RET}_m$.

—*Method call*, $l : b = a_0.u(a_1, \ldots, a_n)$. The processing of a method-call statement depends on whether the callee (method u) is available for analysis or not. For example, when analyzing a Java application, the methods from the JDK library are considered unavailable for analysis. In case of multiple possible callees at a method-call statement (due to virtual dispatch), we take a union of the conditional values generated due to each callee.

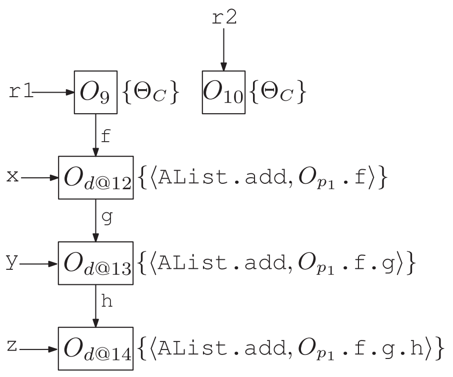
(i) Available-callee (standard, rule not shown). In this case, we first merge the points-to graph G_u at the exit of the called method u into the points-to graph G_m for the caller, using the standard mapping algorithm presented by Whaley and Rinard (1999). For each object O_k added from G_u to G_m while merging, we add the conditional values in $g_u(O_k)$ to $g_m(O_k)$.

(ii) Unavailable-callee. In this case, for each object O_{a_i} pointed-to by the argument a_i, the object(s) reachable from any field of O_{a_i} might change in the method u. Say the object pointed-to by the formal parameter p_i at the entry of u is represented by O_{p_i}. For each field f of O_{a_i}, we denote the dependence of each object $O_x \in G_m(O_{a_i}, f)$ on u by adding a conditional value $\langle u, O_{p_i}.f\rangle$ to $g_m(O_x)$. Note that such conditional values need to be added to all the nodes in the subgraph reachable from O_x as well. We use a macro `extendCVals`$(O_x, \langle u, O_{p_i}.f\rangle)$ that extends the set of conditional values transitively for all the objects reachable from O_x; see Figure 7(b).

Next, say the object O_u represents the objects returned by the method u. After the assignment, to store the dependence of the object pointed-to by b on O_u, we add the conditional value $\langle u, O_u\rangle$ to $g_m(O_{u@l})$, where $O_{u@l} \in \texttt{UCS}_m$. We also set the points-to set of b to a singleton containing $O_{u@l}$.

Irrespective of whether the callee is available or not, we handle the dereference performed by the receiver object a_0 by calling the macro `updateDeref`(O_l, a_0), where $O_l \in \texttt{DRF}_m$.

Note that at a call statement, we need not analyze the callee m if the context c at the call-site is the same as another context c' in which m has already been analyzed. We define a context by applying the idea of *level-summarized value contexts* (Thakur and Nandivada 2019), which is an


r2
r1
O_9 $\{\Theta_C\}$
O_{10} $\{\Theta_C\}$
f
x
$O_{d@12}$ $\{\langle \texttt{AList.add}, O_{p_1}.\texttt{f}\rangle\}$
g
y
$O_{d@13}$ $\{\langle \texttt{AList.add}, O_{p_1}.\texttt{f.g}\rangle\}$
h
z
$O_{d@14}$ $\{\langle \texttt{AList.add}, O_{p_1}.\texttt{f.g.h}\rangle\}$

$\texttt{ALC}_{\texttt{bar}} = \{O_9, O_{10}\}$
$\texttt{OUT}_{\texttt{bar}} = \{O_{d@12}, O_{d@13}, O_{d@14}\}$
$\texttt{DRF}_{\texttt{bar}} = \{O_{11}, O_{12}, O_{13}, O_{14}\}$

$g_{\texttt{bar}}(O_{11}) = \{\Theta_C\}$
$g_{\texttt{bar}}(O_{12}) = \{\Theta_C\}$
$g_{\texttt{bar}}(O_{13}) = \{\langle \texttt{AList.add}, O_{p_1}.\texttt{f}\rangle\}$
$g_{\texttt{bar}}(O_{14}) = \{\langle \texttt{AList.add}, O_{p_1}.\texttt{f.g}\rangle\}$

Fig. 8. $G_{\texttt{bar}}$ and $g_{\texttt{bar}}$ at the exit of the method `bar` shown in Figure 1. For the nodes in $\texttt{ALC}_{\texttt{bar}}$, $\texttt{UCS}_{\texttt{bar}}$, and $\texttt{OUT}_{\texttt{bar}}$, the conditional values are shown next to the node. Redundant conditional values are not shown.

extension of the idea of value contexts (Khedker and Karkare 2008); we consider the set $\{G_m, g_m\}$ as the context at the call-site. We terminate recursion in the standard way, that is by iterating over the strongly-connected components of the call-graph till a fixed-point.

—*Method entry, $m(B\,p_1, \ldots, B\,p_n)$.* The initialization of G_m and g_m at the entry of a method m depends on whether the points-to graph and the partial summary at the call-site c are available or not. We discuss both the cases below.

(i) Available-caller (standard, rule not shown). To construct the points-to graph G_m at the entry of m, for each formal parameter p_i, we add each object O_{a_i} pointed-to by the corresponding actual argument a_i in the points-to graph G_c of the caller c, to $G_m(p_i)$. Next, we copy the subgraph reachable from O_{a_i} in G_c, to G_m. For each object O_x added to G_m in the previous steps, we set $g_m(O_x)$ to $g_c(O_x)$.

(ii) Unavailable-caller. In PACE, the callers of the entry method(s) are not available. For each such method m, we use $\overline{m}$ to denote all its callers at runtime. For each formal parameter p_i of m, we associate an abstract object $O_{p_i} \in \texttt{PAR}_m$, with $g_m(O_{p_i})$ containing the conditional value $\langle \overline{m}, O_{a_i}\rangle$, and add O_{p_i} to the points-to set of p_i. This conditional value indicates the dependence of O_{p_i} on the actual argument O_{a_i} passed to m.

Example. Figure 8 shows the points-to graph $G_{\texttt{bar}}$ and the partial summary $g_{\texttt{bar}}$ at the exit of the method `bar` shown in Figure 1. O_9 and O_{10} are objects allocated at lines 9 and 10, respectively. The statement at line 12 is a load where the object dereferenced via the field `f` is unavailable. We use the abstract object $O_{d@12} \in \texttt{OUT}_{\texttt{bar}}$ to denote the dereferenced object and add an edge from `x` to $O_{d@12}$ to denote that `x` points to $O_{d@12}$. Similarly, the objects $O_{d@13}$ and $O_{d@14}$ are added to $G_{\texttt{bar}}$ at lines 13 and 14, respectively. For each abstract object, Figure 8 also shows the corresponding conditional values in the map $g_{\texttt{bar}}$. The conditional values for O_{11} and O_{12} imply that the corresponding dereferences are guaranteed to be made on concrete objects. The conditional values for O_{13} and O_{14} imply that the objects dereferenced at lines 13 and 14 would be null (or concrete) if the objects pointed-to via O_{p_1}.`f` and O_{p_1}.`f`.`g`, where O_{p_1} is the object pointed-to by the first parameter of `AList.add`, are null (or concrete), respectively.

4.2 Efficient Storage of Partial Summaries

In addition to the pre-apply-meet optimization discussed in Section 3.5, we apply the following three additional optimizations in PACE.

(i) *Store only what is needed.* As discussed in Section 4.1, the domain of program elements for PACE consists of six sets of abstract objects. However, we do not need the analysis information about all the abstract objects for performing null-dereference-check elimination, and, hence, we store only a subset of these sets. For example, for an application method m, it is sufficient to store the conditional values only for the objects in the set $\texttt{DRF}_m$. Similarly, for each library method u, apart from the objects in the set $\texttt{DRF}_u$, we store the conditional values for the elements reachable from the objects in $\texttt{PAR}_u$ and $\texttt{RET}_u$, which may be needed for simplifying the partial summaries of the application methods.

(ii) *Do not store what can be interpreted.* For an element e of a method m, if the set of conditional values $g_m(e)$ is a singleton of the form $\{\Theta_C\}$, we avoid storing the information about e in the .res file. During JIT compilation, if the information about a dereference done at a statement labeled l is missing, we interpret that the dereference is guaranteed to be done on a concrete object. This greatly reduces the number of program elements whose information needs to be printed in the .res files. We could have done the same for elements whose set of conditional values consisted only of Θ_N (in place of Θ_C); however, the space-saving we get would be much lower in practice, considering the small number of dereferenced objects that are guaranteed to be null.

(iii) *Implication.* Say we have a sequence of statements `x = p.f1; y = p.f2`. At runtime, `p` either points to *null* and an exception will be thrown at the first dereference (the second statement will not be executed), or `p` points to a concrete object. In either case, we can elide the null-dereference check at the second statement. We use this observation to further reduce the number of abstract objects created to denote the dereferences in PACE. For example, in the sequence of statements shown above, we do not generate any conditional values for the dereference in the statement `y = p.f2`. This further reduces the size of the stored .res files.

4.3 Results-adapter for PACE

During the JIT compilation of a method m, the summary-simplifier of the fast-precise-analyzer first simplifies the partial summary g_m and generates f_m. Then, for each Bytecode instruction at offset l in m, if the instruction makes a dereference, the results-adapter for PACE checks the value of $f_m(O_l)$. If $f_m(O_l)$ is found to be C (for *Concrete*), the corresponding null-dereference check is elided; else, the existing JVM mechanism is used to proceed with the insertion of the null-dereference check. In Section 7, we show that for a multitude of benchmarks, PACE is able to elide a substantial number of null-dereference checks during JIT compilation, without impacting the JIT compilation time negatively.

5 ESCAPE ANALYSIS FOR SYNCHRONIZATION ELIMINATION IN PYE

We now give an overview of the second analysis that we have implemented in PYE: a top-down context-, flow-, and field-sensitive thread-escape analysis for Java programs (a variation of the approach of Whaley and Rinard (1999); its brief summary can be found in Section 2). We use the results of this analysis to elide the acquire/release synchronization operations associated with objects that do not escape their allocating thread. We call this instantiation the Escape Analysis for Synchronization Elimination (or EASE, in short).

As discussed in Section 3.6, in order to perform an analysis using PYE, we need to specify the domain $\mathcal{D}$, the lattice formed by `Val`, the processing of each relevant statement by the partial-analyzer, and the results-adapter. We now describe the same for EASE.

5.1 Partial-analyzer for EASE

Figure 5 shows the domain $\mathcal{D}_{ease}$ and the lattice $\texttt{Val}_{ease}$ for EASE. For each method m, $\mathcal{D}_{ease}$ consists of six sets of abstract objects: (i) $\texttt{ALC}_m$: one object O_l per allocation statement labeled l; (ii) $\texttt{PAR}_m$: one object O_{p_i} representing the objects pointed-to by the parameter p_i; (iii) $\texttt{RET}_m$: all the objects returned by m; (iv) $\texttt{OUT}_m$: one object $O_{d@l}$ for an unavailable dereference at a statement

$$
\begin{array}{rll}
\text{[allocation]} & l : a = \mathit{new}\ B(); & \begin{cases} \text{if } (\mathit{isMultiThreaded}(B)) \text{ then} \\ \quad g_m[O_l \leftarrow \{\Theta_E\}] \\ \text{else} \\ \quad g_m[O_l \leftarrow \{\Theta_D\}] \end{cases} \\
\text{[store]} & a.f = b; & \begin{cases} \text{if } (\mathit{isStatic}(f)) \text{ then} \\ \quad \forall O_x \in G_m.\mathit{reachables}(b) \\ \qquad g_m[O_x \leftarrow \{\Theta_E\}] \\ \text{else} \\ \quad \forall O_x \in G_m.\mathit{reachables}(b) \\ \qquad g_m[O_x \cup\!\!\leftarrow g_m(O_a)] \end{cases} \\
\text{[load]} & l : a = b.f; & \begin{cases} \text{if } (\exists O_b \in G_m(b),\ s.t.\ G_m(O_b, f) = \emptyset) \text{ then} \\ \quad \forall \langle u, x\rangle \in g_m(O_b) \cap \mathtt{DCVal} \\ \qquad g_m[O_{d@l} \cup\!\!\leftarrow \{\langle u, x\rangle, \langle u, x.f\rangle\}] \end{cases} \\
\text{[synchronization]} & l : \mathit{synchronized}(a) & \begin{cases} g_m[O_l \leftarrow \cup_{\forall O_a \in G_m(a)} g_m(O_a)] \end{cases} \\
\text{[unavailable-callee]} & l : b = a_0.u(a_1, ..., a_n); & \begin{cases} \forall O_{a_i} \in G_m(a_i) \\ \quad \mathtt{extendCVals}(O_{a_i}, \langle u, O_{p_i}\rangle) \\ g_m[O_{u@l} \leftarrow \{\langle u, O_u\rangle\}] \end{cases}
\end{array}
$$

Fig. 9. Partial-analysis rules for EASE. The updates to the points-to graph G_m, and the rules [unavailable-caller] and [return], are similar to those for PACE (see Figure 6), and, hence, skipped.

labeled l; (v) $\mathtt{UCS}_m$: one object $O_{u@l}$ per unavailable method u at a call-statement labeled l; and (vi) $\mathtt{SYN}_m$: one object O_l per synchronization statement labeled l. The set $\mathtt{Val}_{ease}$ forms a lattice with two elements: *DoesNotEscape* (D or $\top$) and *Escapes* (E or $\bot$). The corresponding conditional values are Θ_D and Θ_E, respectively. The definition of the meet ($\sqcap$) operation is standard. At runtime, for each method m, the goal of EASE is to compute the *escape-status* (D or E) of each object in $\mathtt{SYN}_m$.

Figure 9 shows how the proposed partial-analyzer of EASE processes each statement that may affect the partial summary g_m for the method m being analyzed. As the rules for maintaining the points-to graph G_m and the processing of method entry and return statements are similar to those for PACE (see Section 4.1), we skip showing/discussing the same. Similar to PACE, we assume that each statement has a unique label associated with it.

—*Allocation, $l : a = new\, B()$.* At an allocation statement $a = new\, B()$ at line l, we use the abstract object $O_l \in \mathtt{ALC}_m$ to represent the object(s) allocated at l. The conditional value associated with O_l is either Θ_E or Θ_D, depending on whether B is a *multi-threaded* class or not. We term a class as multi-threaded if it is a subclass (immediate or otherwise) of `java.lang.Thread` or implements `java.lang.Runnable`.

—*Store, $a.f = b$.* At a statement of the form $a.f = b$, if f is a static field, for each object O_x reachable from b, we set $g_m(O_x)$ to the singleton Θ_E, indicating that O_x now escapes; else, we indicate the dependence of the escape-status of O_x on that of O_a by adding the conditional values in $g_m(O_a)$ to $g_m(O_x)$.

—*Load, $l : a = b.f$.* Similar to PACE, for an object $O_b \in G_m(b)$, if $g_m(O_b)$ consists of a dependent conditional value $\langle u, x\rangle$, it implies that the escape-status of O_b depends on the program element x in method u. In such a case, the object(s) pointed-to by $O_b.f$ might change/escape in u. To denote this dependence, we add the abstract object $O_{d@l} \in \mathtt{OUT}_m$ to $G_m(O_b, f)$, and add the conditional values $\langle u, x\rangle$ and $\langle u, x.f\rangle$ to $g_m(O_{d@l})$. These conditional values indicate that $O_{d@l}$ escapes if either of the program elements x or $x.f$ in the method u escape.

$g_{\texttt{foo}}(O_3) = \{\langle \texttt{AList.add}, O_{p_1}\rangle\}$
$g_{\texttt{foo}}(O_4) = \{\Theta_E\}$
$g_{\texttt{foo}}(O_7) = \{\langle \texttt{AList.add}, O_{p_1}\rangle, \langle \texttt{AList.add}, O_{p_1}.\texttt{f}\rangle\}$
$g_{\texttt{foo}}(O_8) = \{\langle \texttt{AList.add}, O_{p_0}\rangle\}$
$g_{\texttt{foo}}(O_5) = \{\Theta_D\}$
$g_{\texttt{foo}}(O_{d@10}) = g_{\texttt{foo}}(O_{11}) = \{\langle \texttt{AList.add}, O_{p_1}\rangle, \langle \texttt{AList.add}, O_{p_1}.\texttt{f}\rangle\}$

$\texttt{ALC}_{\texttt{foo}} = \{O_3, O_4, O_7, O_8\}$
$\texttt{SYN}_{\texttt{foo}} = \{O_5, O_{11}\}$
$\texttt{OUT}_{\texttt{foo}} = \{O_{d@10}\}$

Fig. 10. The conditional values in $g_{\texttt{foo}}$ for the method foo in Figure 2(a). Redundant conditional values are not shown.

—*Synchronization, $l : synchronized(a)$.* Here, we use the abstract object $O_l \in \texttt{SYN}_m$ to represent the synchronization statement labeled l. For each object O_a in the points-to set of a, we add the conditional values in $g_m(O_a)$ to $g_m(O_l)$. This indicates that the synchronization operation at l can be elided if none of the objects in the points-to set of a at l escape.

—*Unavailable-callee, $l : b = a_0.u(a_1, \ldots, a_n)$.* The handling of a method call where the callee u is unavailable is similar to that for PACE except for a minor difference. Here, for each argument a_i, even the top-level object $O_{a_i} \in G_m(a_i)$ may escape in u (if it is assigned to a static field in u, for example). We indicate the dependence by adding the conditional value $\langle u, O_{p_i}\rangle$ to $g_m(O_{a_i})$, indicating that O_{a_i} might escape if the object O_{p_i}, representing the object pointed-to by the formal parameter p_i at the entry of u, escapes. Similarly, all the objects reachable from O_{a_i} in G_m also depend on u. We call the macro $\texttt{extendCVals}(O_{a_i}, \langle u, O_{p_i}\rangle)$ to add the corresponding conditional values (see Figure 7(b)).

Example. Figure 10 shows the conditional values generated by the partial-analyzer of EASE as seen at the exit of the method `foo` shown in Figure 2(a). As the object O_3 pointed-to by the variable `r1` is passed to the unavailable method `AList.add`, the conditional value in $g_{\texttt{foo}}(O_3)$ indicates that the escape-status of O_3 depends on the escape-status of the object O_{p_1} pointed-to by the first formal parameter at the entry of `AList.add`. Similar is the case for the object O_8. As the object O_4 becomes reachable from the static variable `global` at line 6, $g_{\texttt{foo}}(O_4)$ consists of the conditional value Θ_E, implying that O_4 escapes. However, as we separately keep track of abstract objects in the set $\texttt{SYN}_{\texttt{foo}}$, we are able to capture the fact that O_4 does not escape at line 5, and, hence, $g_{\texttt{foo}}(O_5)$ consists of the conditional value Θ_D. This enables the elision of the synchronization at line 5 (at runtime). The conditional values for the synchronization statement at line 11 indicate that the corresponding synchronization operation can be elided if the object O_{p_1} pointed-to by the formal parameter p_1 and the object pointed-to by $O_{p_1}.\texttt{f}$ do not escape in `AList.add`.

Synchronized methods. In Java, apart from synchronized statements, methods can also be declared as *synchronized* to indicate that any code in those methods cannot be executed by concurrent threads. If an instance method is declared as synchronized, it is equivalent to a synchronized statement on the receiver object, enclosing the body of the method. If the synchronized method is static, the synchronization operation is performed on the global object associated with its declaring class. Similar to the approach used by Lee and Midkiff (2006), we elide the synchronization operation associated with such a method, if the method is never called in a sequence of calls originating from a `run` method of a multithreaded class. For both these cases, we store the conditional values in a special abstract object associated with the corresponding method.

5.2 Efficient Storage of Partial Summaries

In addition to the pre-apply-meet optimization discussed in Section 3.5, we apply the following two additional optimizations in EASE.

(i) *Store only what is needed.* For an application method m, we store the conditional values only for the objects in the set $\texttt{SYN}_m$. For each library method u, apart from the objects in the set $\texttt{SYN}_u$, we also store the conditional values for the elements reachable from the objects in $\texttt{PAR}_u$ and $\texttt{RET}_u$.

(ii) *Do not store what can be interpreted.* For an element e of a method m, if the set $g_m(e)$ of conditional values is a singleton of the form $\{\Theta_D\}$, we avoid storing the information about e in the .res file. During JIT compilation, if the information about an abstract object ($\in \texttt{SYN}_m$) is missing, we interpret that the associated synchronization operation can be safely elided.

5.3 Results-adapter for EASE

To perform synchronization elimination in the HotSpot JVM, the optimizer needs to know whether the object associated with a synchronization statement does-not-escape and, consequently, if the synchronization operation can be eliminated. Our results-adapter reads the escape-status of the abstract object corresponding to each of the synchronization statements and sets a field `isEliminatable` in the synchronization statement accordingly.

Example. The conditional values generated by the partial-analyzer for the method `AList.add` (see Figure 1) include the dependences listed below; recall that for each method m with unavailable-caller, we use $\overline{m}$ to denote its callers at runtime.

$$g_{\texttt{AList.add}}(O_{p_0}) = \{\langle\overline{\texttt{AList.add}}, O_{a_0}\rangle\}$$
$$g_{\texttt{AList.add}}(O_{p_1}) = \{\langle\overline{\texttt{AList.add}}, O_{a_0}\rangle, \langle\overline{\texttt{AList.add}}, O_{a_1}\rangle\}$$

The conditional values $\langle\texttt{AList.add}, O_{a_0}\rangle$ and $\langle\texttt{AList.add}, O_{a_1}\rangle$ are added to the map $g_{\texttt{AList.add}}$ for O_{p_0} and O_{p_1}, respectively, at the entry of the method `AList.add`. When the object O_{p_1} is stored into the array pointed-to by $O_{p_0}.\texttt{f}$ (at line 9), the conditional values in $g_{\texttt{AList.add}}(O_{p_0})$ are added to $g_{\texttt{AList.add}}(O_{p_1})$.

During JIT compilation of the application method `foo`, the summary-simplifier evaluates the conditional values (see Figure 10) for the synchronization objects of lines 5 and 11 (O_5 and O_{11}, respectively). For O_5, the conditional value Θ_D simply evaluates to the value $D \in \texttt{Val}_{ease}$. To simplify the conditional values for O_{11}, our algorithm starts with a list L of conditional values $\{\langle\texttt{AList.add}, O_{p_1}\rangle, \langle\texttt{AList.add}, O_{p_1}.\texttt{f}\rangle\}$ (given by $g_{\texttt{foo}}(O_{11})$ shown in Figure 10). Simplifying these conditional values adds the elements of the set $g_{\texttt{AList.add}}(O_{p_1})$ to L. As O_8 and O_3 are the actual arguments O_{a_0} and O_{a_1}, respectively, our algorithm adds the elements of $g_{\texttt{foo}}(O_8)$ and $g_{\texttt{foo}}(O_3)$ to L to obtain: $L = \{\langle\texttt{AList.add}, O_{p_1}\rangle, \langle\texttt{AList.add}, O_{p_1}.\texttt{f}\rangle, \langle\overline{\texttt{AList.add}}, O_{a_0}\rangle, \langle\overline{\texttt{AList.add}}, O_{a_1}\rangle, \langle\texttt{AList.add}, O_{p_0}\rangle\ \}$.

At this stage, no further simplification is possible and we reach a fixed-point. As mentioned in Section 3.4, after attaining the fixed-point, for each conditional value $\mathcal{T}_i$ in the worklist, we set $[\![\mathcal{T}_i]\!]$ to D (the top of the lattice). Thus, the summary-simplifier generates the escape-status: $f_{\texttt{foo}}(O_5) = f_{\texttt{foo}}(O_{11}) = D$. Thereby the results-adapter sets the `isEliminatable` field for the synchronization statements at lines 5 and 11 to `true`. This field is used by the following pass of synchronization elimination to perform the necessary optimization.

Note that a fully static (flow-sensitive) analysis would be able to elide the synchronization operation only at line 5. On the other hand, the C2 compiler of the HotSpot JVM performs a connection-graph (Choi et al. 1999) based partially-interprocedural escape analysis during JIT compilation, which can elide the synchronization operation at line 11, but not at line 5. Using PYE, and by maintaining abstract objects separately in $\texttt{SYN}_{\texttt{foo}}$, EASE is able to elide the synchronization operations both at lines 5 and 11. We show in Section 7 that the overheads involved for achieving this precision during JIT compilation are quite small, in fact, less than the existing analysis performed by C2.

6 DISCUSSION

In this section, we first discuss some subtle aspects in the current design of PYE, followed by its correctness argument.

1. Deoptimization. PYE handles deoptimization scenarios (for example, because of dynamic classloading, failed speculative type-checks, and so on) in a natural manner: the set of new and old methods to be re-compiled are obtained by analyzing the call graph and optimized using their partial summaries (new or existing). This set can be made further precise using a scheme similar to that of Cooper et al. (1986).

2. Callbacks. PYE analyzes each library installation independent of the application. Consequently, if a library method m has a callback to a method in the application program, the called method cannot be analyzed in this context. In such a scenario, we compile m (and the dependent methods) with the existing analyses built in the JVM, and not with the analysis-results generated by PYE. Since callbacks are used infrequently in practice (during our evaluation over real-world benchmarks, we have not found any case where we lose precision because of this design choice), we believe it to be an acceptable limitation of PYE. There could be multiple ways to handle callbacks more precisely. For example, we could perform a fast (perhaps imprecise) analysis to find out whether the called-back method may indeed affect the results for the given analysis and fall-back only if it does. Another way to handle callbacks is to statically create *gaps* (Arzt and Bodden 2016) at call-sites that may be involved in a callback, and fill these gaps with more precise information during JIT compilation. We leave the integration of the techniques of Arzt and Bodden into PYE as a future work.

3. Verification. The summaries (generated by the partial-analyzer) transferred along with the class files to the target machine may get corrupted (intentionally, or otherwise). Consequently, the fast-precise-analyzer may derive wrong analysis-results. Currently, we resolve this issue by using public-key cryptography (Stinson 1995). However, keeping in mind its limitations (for example, trusting the partial-analyzer), we are working on a fast verifier (similar to Java Bytecode type-checking) to validate the results in the JVM itself.

4. Heap abstraction. We abstract all the elements of an array in a field-insensitive manner, which leads to well understood imprecision. Consider the methods `baz` of class `B` and `first` of class `AList`, as shown in Figure 11, in the context of PACE. For the dereference at line 8, the set of conditional values generated by the partial-analyzer would be: $\{\langle \texttt{AList.add}, O_{\texttt{AList.add}} \rangle\}$. However, as the array `arr` does not distinguish among its various elements (it is standard to treat arrays field-insensitively for scalability), the method `first` would conservatively generate the value $\{\Theta_N\}$ for its return-value, and, hence, the dereference during JIT compilation would not be elided. Similarly, the abstraction of objects by their allocation site (that is, no heap-cloning (Nystrom et al. 2004)) leads to understandable imprecision. For example, in Figure 11(b), the partial-analyzer finds that at line 6, `r2` may point to `null` and hence cannot elide the null-check. We believe it would be an interesting future work to extend PYE to support more precise forms of heap abstractions.

5. Conditional values. The conditions used in the conditional values depend on the specific analysis being performed. For example, for EASE, the conditions are based on the escape-information, leading to conditional values such as x escapes if y escapes. In contrast, for implementing the taint analysis by Arzt and Bodden (2016), each "source" of the taint may be treated as a tainted object, and the conditions may be based on points-to/alias relationships, leading to conditional values such as x points to a tainted object z', if y points to z. These points-to conditions can be on the argument-objects passed and the return values of the unavailable methods. Importantly, note that such variations (naturally, analysis-dependent) still fit into the general two-pronged approach presented as part of PYE.

```
class B { ... // from Figure 1
  void baz() {
    B r1 = new B();
    List l = new AList();
    l.add(r1);
    l.add(null);
    X x = l.first();
    Y y = x.g;
  }
}
```

```
class AList<E> { ... // from Figure 1
  E first() {
    return arr[0];
  }
}
```

(a)

```
class C {
  void foo() {
    X r1 = new X();
    bar(null);
    X r2 = bar(r1);
    ... = r2.g;
  }
  void bar(X rx) {
    Y y1 = new Y();
    y1.f = rx;
    return y1.f;
  }
}
```

(b)

Fig. 11. Methods to illustrate some issues in heap abstraction.

6. Levels of granularity. Though we discuss PYE at method-level granularity, these ideas can also be extended to other levels of granularity (for example, storing/simplifying summaries per class, package, and so on) without impacting the precision. The choice of the appropriate granularity levels can lead to interesting tradeoffs: while storing/reading/simplifying summaries at higher levels of granularity can amortize (and speedup) the overall disk reads, it could also lead to increased overheads from reading/simplifying summaries of even those methods that may not be compiled.

6.1 PYE: Correctness

It may be noted that the precision of an analysis implemented in PYE depends upon the algorithm used in the partial-analyzer to generate the partial summaries. Thus, PYE can be thought to be parametric on the analysis algorithm. If we represent the analysis being performed as Ψ, then the PYE variation of Ψ can be denoted as PYE(Ψ). We now state the correctness theorem for our proposed approach for programs that may call library methods that, in turn, invoke no callbacks. Later, we extend the argument to library methods that may invoke callbacks.

Definition 6.1. The set $\{\langle y_1, v_1^1, v_2^1\rangle, \langle y_2, v_1^2, v_2^2\rangle, \cdots\}$ of conditional values generated by PYE(Ψ) for a variable x, for any program P at a program point L, is considered to be "correct" if during the whole-program analysis the following conditions hold: (i) $\exists$ a set S of integers, such that $\forall i \in S$, Ψ computes the value of y_i to be v_1^i at L; and (ii) Ψ computes the dataflow value of x to be $\sqcap_{\forall i \in S} v_2^i$. That is, a correct set of conditional values includes all the dependencies and nothing more.

Theorem 6.2. *Given a whole-program analysis algorithm Ψ, for any program P, the analysis-results obtained using PYE(Ψ) for the program elements of P will match those obtained using Ψ. That is, if $\mathcal{D}$ is the set of program elements of P, then after simplification of partial summaries, $\forall x \in \mathcal{D}$, PYE($\Psi$)($x$) = Ψ(x).*

Proof. *(Proof Sketch)*

We prove the theorem by contradiction. Say there exists a whole-program analysis Ψ and a program element x in method m such that at some program point L, $\text{PYE}(\Psi)(x) = s_1$ and $\Psi(x) = s_2$, and $s_1 \neq s_2$.

This implies that while Ψ has found the dataflow value of x at L (the most precise solution) to be s_2, the summary-simplifier has found the dataflow value of x at L to be s_1. There can be two cases under which the summary-simplifier can find the dataflow value of x at L to be s_1 ($\neq s_2$):

(i) In the set of summaries provided by the partial-results-accumulator, at program point L, $g_m(x)$ was a singleton containing s_1 (a simple conditional value). This implies that the partial-results-accumulator has incorrectly obtained the dataflow value as s_1 from the partial-analyzer. But as the partial-analyzer implements Ψ for performing the static analysis and gives a singleton with a simple conditional value only for objects that do not depend on any library calls, Ψ would also find the dataflow value of x as s_1 (and, hence, $s_1 = s_2$). A contradiction.

(ii) In the set of summaries provided by the partial-results-accumulator, at program point L, $g_m(x)$ consists of a set of dependent conditional values, and $g_m(x)$ was simplified as s_1 by the summary-simplifier. There are two sub-cases:

(a) the partial-analyzer generated a correct set of conditional values for x, but the summary-simplifier generated an imprecise solution. As discussed in Section 3.4, the summary-simplifier iteratively solves the dependent conditional values till no more of them can be simplified further, which generates the most precise solution for the *given* set of conditional values. A contradiction.

(b) The partial-analyzer generated an incorrect set of conditional values for x. Note that the partial-analyzer is an implementation of Ψ with the only difference being in the output for the ones related to the conditional values such that $\forall x \in \mathcal{D}$ at each program point L. If PYE$(\Psi)(x)$ is a not a simple conditional value, then the partial-analyzer adds all the required dependent conditional values (and only those) that denote a dependence of $\Psi(x)$ on the unavailable parts of P. That is, PYE marks all the required dependences and nothing more. That is, as per Definition 6.1, the summary generated by the partial-analyzer (PYE$(\Psi(x))$) is correct. A contradiction. □

Theorem 6.2 guarantees that during JIT compilation, a whole program analysis Ψ can be equivalently replaced by PYE(Ψ), when the called library methods do not, in turn, invoke callbacks. Even in cases where the library methods may invoke callbacks, extending PYE with techniques proposed by Arzt and Bodden (2016) can lead to similar precision. However, when using our above discussed conservative scheme to handle callbacks, the Theorem 6.2 statement can be modified to state that after the simplification of partial summaries, $\forall x \in \mathcal{D}$, PYE$(\Psi)(x) \sqsubseteq \Psi(x)$, where the relation $x \sqsubseteq y \Rightarrow x \sqcap y = x$.

Corollary 6.3. *In a JIT compiler, a semantics-preserving optimization based on a whole program analysis Ψ can use PYE(Ψ) instead and still remain semantics-preserving.*

Proof. Proof follows directly from Theorem 6.2 and the above discussion thereof. □

7 IMPLEMENTATION AND EVALUATION

We implemented the PYE framework (see Figure 3(b)) in two parts: (i) interfacing of the partial-analyzer in the Soot optimization framework (Vallée-Rai et al. 1999) version 2.5.0—approximately 2,000 lines of code; (ii) different components of the fast-precise-analyzer in the HotSpot Server Compiler (C2) of the OpenJDK HotSpot JVM (Paleczny et al. 2001) version 7—approximately 1,000 lines of code. To understand the usability of PYE, we used PYE to instantiate PACE (Section 4) and EASE (Section 5). The associated respective partial-analyzers consist of about 4,100 and 4,000 lines of Java code (in Soot), and the respective results-adapters consist of about 550 and 250 lines of C++ code (in C2). In addition, we use the extremely helpful tool TamiFlex (Bodden et al. 2011) version 2.0.3 to eliminate the reflection-based code from the original benchmarks so that they can be analyzed by Soot.

S.No.	Benchmark	Number of class files	.class size (MB)	.res size (MB)		Overhead (%)		`DCVal` (%)	
				PACE	EASE	PACE	EASE	PACE	EASE
1.	aes	297	2.1	0.24	0.16	11.33	7.38	42.97	10.0
2.	compiler	306	2.1	0.24	0.15	11.29	7.33	42.96	10.0
3.	compress	308	2.1	0.24	0.16	11.19	7.48	42.59	10.0
4.	fft	301	2.1	0.24	0.16	11.29	7.38	43.72	9.09
5.	lu	300	2.1	0.24	0.16	11.33	7.38	43.38	9.09
6.	montecarlo	300	2.1	0.23	0.16	11.14	7.38	43.50	9.09
7.	rsa	297	2.1	0.30	0.16	11.05	7.38	43.30	10.0
8.	signverify	297	2.1	0.24	0.16	11.29	7.38	43.39	10.0
9.	sor	301	2.1	0.24	0.16	11.19	7.38	42.51	9.09
10.	sparse	300	2.1	0.24	0.16	11.19	7.38	42.68	9.09
11.	sunflow	406	2.7	0.32	0.22	11.85	8.26	35.55	4.76
12.	avrora	527	2.6	0.04	0.02	1.54	0.85	62.55	0.0
13.	eclipse	1344	10	1.50	0.84	15.0	8.42	18.17	5.05
14.	fop	1027	5.8	0.36	0.20	6.21	3.38	35.51	11.11
15.	h2	324	2.2	0.04	0.02	1.91	1.05	62.78	0.0
16.	luindex	200	1.3	0.04	0.03	2.92	1.85	61.00	100.0
17.	lusearch	198	1.2	0.04	0.03	3.33	2.08	59.54	100.0
18.	pmd	688	3.7	0.04	0.03	1.19	0.68	56.16	0.0
19.	xalan	638	3.7	0.04	0.03	1.19	0.78	49.74	100.0
20.	moldyn	14	0.06	0.003	0.002	5.33	2.67	52.2	0.0
21.	raytracer	22	0.09	0.005	0.004	5.76	3.91	88.8	0.0
22.	specjbb	82	0.51	0.07	0.03	12.99	6.69	73.96	12.27
	GeoMean	268	1.7	0.11	0.07	6.41	3.96	47.27	7.1

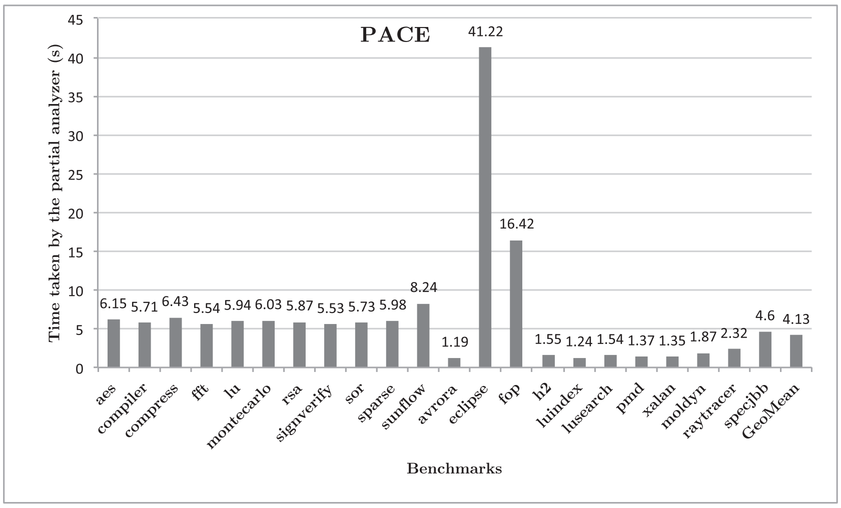
Fig. 12. Details of the benchmarks used, storage overhead for .res files, and the percentage of elements with dependent conditional values (`DCVal`) in the generated .res files.

7.1 Evaluation Setup

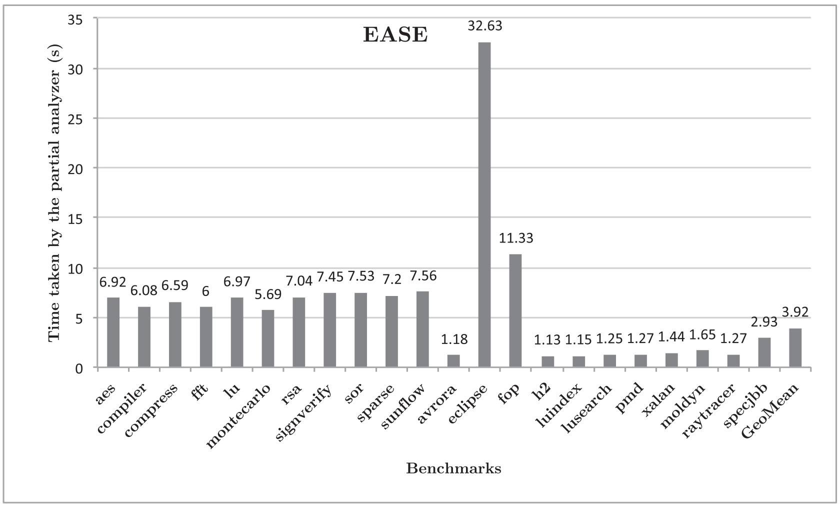
We evaluated PYE and its two instantiations on 22 benchmarks from 4 benchmark suites: (i) AES, COMPILER, COMPRESS, FFT, LU, MONTECARLO, RSA, SIGNVERIFY, SOR, SPARSE, and SUNFLOW from SPECjvm (2008)—using the "default" input; (ii) AVRORA, ECLIPSE, FOP, H2, LUINDEX, LUSEARCH, PMD, and XALAN from the DaCapo suite (Blackburn et al. 2006) version 9.12—using the "default" input; (iii) MOLDYN and RAYTRACER from Section C of the JGF suite (Daly et al. 2001)—using "SizeB" inputs; and (iv) SPECjbb (2005)—using the default 30 seconds ramp-up time and 240 seconds measurement window. In case of BATIK (from DaCapo), we could not run the original program on our system for the "default" input (threw *TruncatedFileException*). The rest of the benchmarks excluded from the original DaCapo and SPECjvm suites could not be analyzed—either by Soot or by TamiFlex. Our experiments have been performed on a 2.3GHz AMD Abu Dhabi system (DVFS disabled) with 64 cores and 512GB of memory, running CentOS 6.4.

Figure 12 lists some static characteristics about the benchmarks included in the study. The sizes of the benchmarks listed in Figure 12 varied from 60KB (small programs) to 10MB (large applications), and the number of class files varied from 14 to 1,344. In the SPECjvm suite, it can be seen that the benchmarks from the same group have similar sizes; for example, AES, RSA, and SIGNVERIFY from the CRYPTO group, and FFT, LU, MONTECARLO, SOR, and SPARSE from the SCIMARK group. This similarity occurs because most of the code among the benchmarks in the same group is shared (for example, the harness and the utilities).

We now present an evaluation to study the impact of our proposed framework PYE. We divide the evaluation into two parts: (i) evaluation related to the static-compilation time analysis



(a)



(b)

Fig. 13. Time taken (in seconds) by the partial analyzers of (a) PACE and (b) EASE.

(involves the partial-analyzer of PYE); and (ii) evaluation related to the JIT-compilation time analysis (involves the fast-precise-analyzer of PYE).

7.2 Evaluation of the Partial-analyzer

In this section, we evaluate the partial-analyzers for PACE and EASE by focusing on the time taken to compute the partial summaries, the storage overhead of the generated .res files, and the precision of the conditional values in the .res files.

7.2.1 Time Taken by the Partial-analyzer. Figure 13 shows the time taken by the partial-analyzer for all the benchmarks. On average, the partial-analyzer for PACE took 4.13 seconds across all the benchmarks, and that for EASE took 3.92 seconds across all the benchmarks. We observe that the time required is mainly dependent on the size of the individual benchmark: less time for smaller benchmarks (for example, JGF) and more time for larger DaCapo benchmarks (for example, ECLIPSE and FOP). Considering that the analyses performed by PACE and EASE are very precise (context, flow, and field-sensitive), we argue that the analysis time is quite reasonable. Further, this analysis time (once per static-compilation) does not get added to the time for final execution (may happen many times).

7.2.2 Storage Overhead. The summaries generated by the partial-analyzer for each benchmark are stored in plain text as a file `<benchName>.<analysis>.res`. These summaries are piggy-backed with the class-files of the benchmark and transferred to the JVM in which the benchmark is executed. Figure 12 lists the sizes of the .res files (in MB) for each benchmark, for both PACE and EASE. It is evident that these files are very small (110KB and 70KB on average, respectively, for PACE and EASE). As compared to the sizes of the corresponding benchmarks, the average storage overheads for PACE and EASE are 6.41% and 3.96%, respectively—arguably quite low.

Lee and Midkiff (2006) had proposed a two-phase escape analysis for the Jikes RVM (Alpern et al. 2005). They compute connection graphs (a representation similar to points-to graphs) for different methods offline and merge the connection graphs to complete an interprocedural analysis during JIT compilation. Compared to the overhead reported by Lee and Midkiff for storing the connection graphs (68% over the class files), the storage overhead for partial summaries in EASE is quite low (only 3.96% over the class files).

7.2.3 Precision of Conditional Values. For a program element e in method m, the conditional values in $g_m(e)$ may be either simple or dependent (see Section 3.3). If all the conditional values in $g_m(e)$ are simple, it implies that the analysis-result $f_m(e)$ for e does not depend on any other element. For the rest of the elements, the dependent conditional values need to be evaluated in the fast-precise-analyzer (at runtime). The last column of Figure 12 shows the percentage of stored elements for which the partial summary consists of at least one dependent value. On average, the analysis-results for 47.27% and 7.1% of the elements for PACE and EASE, respectively, consist of dependent conditional values, and hence cannot be computed precisely using any static whole-program analysis that handles library calls conservatively (alternative A1 in Section 1). These numbers show that the potentially achievable precision for PACE and EASE over a static whole-program analysis is significant.

7.3 Evaluation of the Fast-precise-analyzer

In this section, we evaluate the fast-precise-analyzers for PACE and EASE by focusing on the precision of the generated analysis-results and the time taken during JIT compilation, compared to the existing state-of-the-art analyses in the C2 compiler of the HotSpot JVM version 7. The goal of this comparison is to demonstrate that PYE leads to the generation of more precise analysis-results, while spending time similar to (and in some cases lower than) the existing imprecise analyzers of C2. For each of the analyses, we evaluate the fast-precise-analyzer in the default setting of the HotSpot JVM (called the *mixed* mode) that uses an interpreter, the client C1 compiler, and the server C2 compiler. In this mode, the C2 compiler is invoked only for those methods/loops that are invoked/iterated more than a threshold number of times (usually 10,000–15,000).

7.3.1 Precision of Generated Results. For statements that dereference an object, the JVM needs to perform a null-dereference check—if the dereferenced object is null, then the check fails and a *NullPointerException* is thrown. The C2 compiler applies the implication optimization discussed in Section 4.2 to avoid inserting several checks; however, the rest of the checks remain explicit and need to be performed during program execution. Figure 14 compares the number of explicit

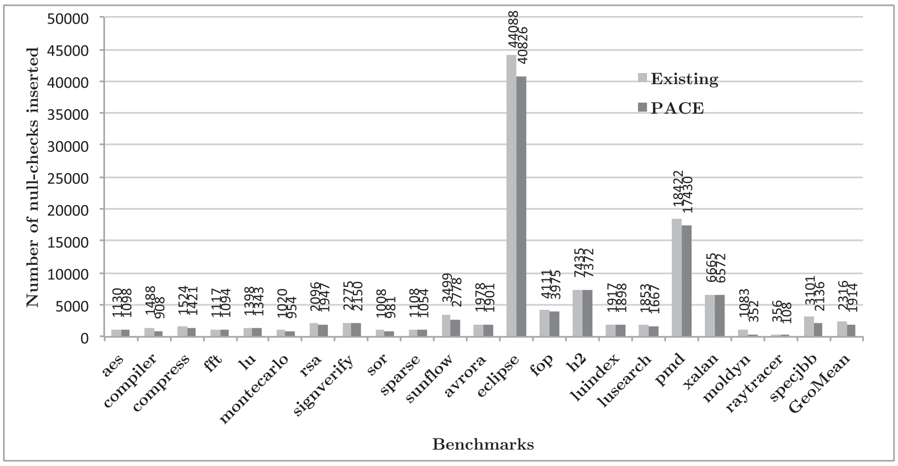
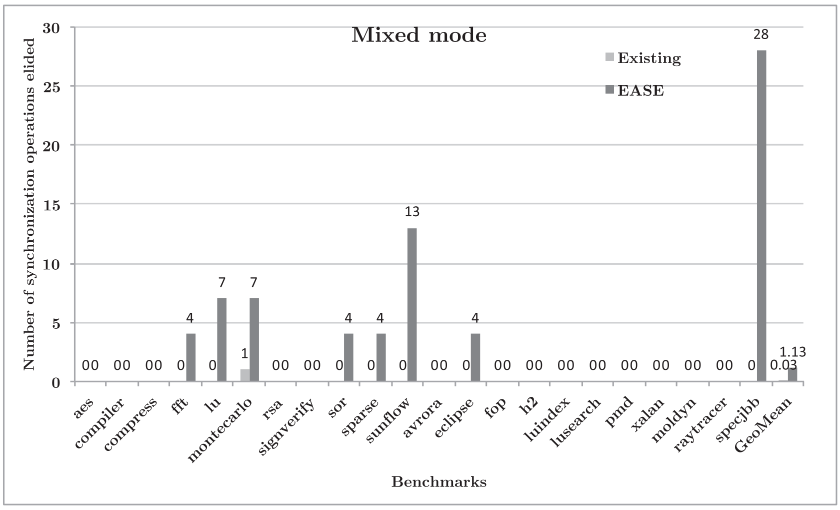


Fig. 14. Number of explicit null-checks inserted by the existing analyzer of the C2 compiler and PACE; the smaller the better.

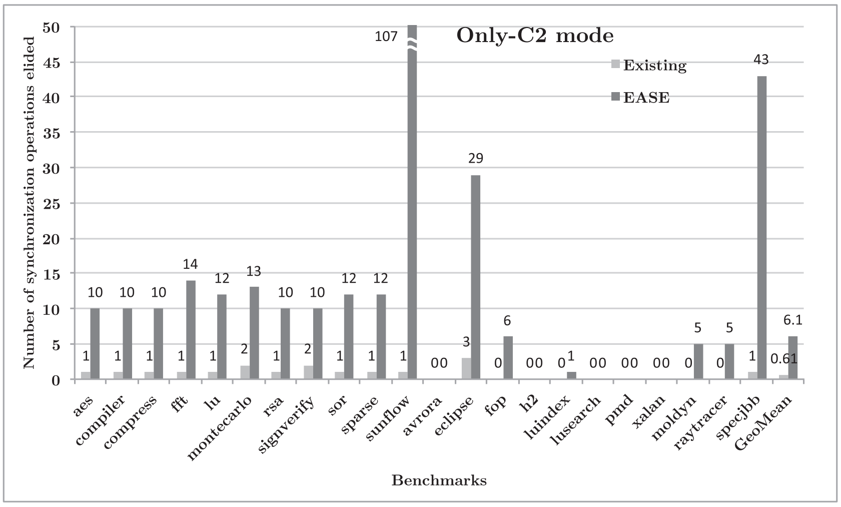
null-dereference checks inserted by PACE, with that by the existing analyzer of C2. As evident, the number of null-dereference checks inserted by PACE is significantly lower than that by the existing analyzer of C2—17.36% on average. This clearly demonstrates the enhanced precision achieved by PACE. In Section 7.3.2, we show that the cost to achieve this precision is negligibly low.

In Java programs, the synchronization statement `synchronized(a) {S}` can be used to execute the statement *S* in a mutually exclusive manner. The JVM implements mutual exclusion by acquiring the lock associated with the object pointed-to by the variable a. Based on the work of Choi et al. (1999), the C2 compiler performs a partially interprocedural escape analysis: the connection-graphs are intraprocedural, but a Bytecode-level flow-insensitive analysis is performed at method call-sites. This analysis is used to identify and elide the synchronization operations for which the associated object is guaranteed to be accessed by a single thread. Figure 15(a) compares the number of synchronization operations elided by EASE with that by the existing escape analyzer of C2. We can see that during JIT compilation, while the existing analyzer of C2 does not elide any synchronization operation across most of the benchmarks (except MONTECARLO), the precise nature of EASE leads to the elision of a significant number of synchronization operations in many benchmarks (up to 28 elisions, in case of SPECJBB). In Section 7.3.2, we show that the time to obtain this precision is much less compared to the time spent in performing escape analysis in the existing C2 compiler.

Note that the low number of elisions is due to two reasons: (i) In most of the programs, the synchronization constructs are indeed necessary. (ii) The methods containing the synchronization statements are not compiled by C2 because of the high threshold limit. As a side study, to get an estimate on the upper limit on the number of synchronization operations that could be elided for long running programs, where more methods may get compiled by C2, we evaluated EASE and the existing analyzer of C2 in an *only-C2* mode. Here, we disabled the interpreter and the C1 compiler, and compiled every method using C2. Figure 15(b) compares the number of synchronization operations elided by EASE with that by the existing analyzer in the only-C2 mode. It can be seen that compared to Figure 15(a), the existing analyzer of C2 finds opportunities (albeit small in number) for synchronization elimination in more benchmarks. In contrast, EASE is able to find many more opportunities (up to 107). This shows that for long running programs, EASE may lead to the elision of more synchronization operations than the existing analyzer of C2.



(a)



(b)

Fig. 15. Number of synchronization operations elided by the existing analyzer of the C2 compiler and that by EASE: (a) in mixed mode and (b) in only-C2 mode; the larger the better.

7.3.2 Time Taken During JIT Compilation. We now evaluate the time taken by the fast-precise-analyzers of PACE and EASE during JIT compilation. For reading the .res files, we spawn a separate thread for the partial-results-accumulator during the initialization of the JVM, and wait for the thread to terminate before using the results in the C2 compiler. The time to read the .res files varies with the size of the .res files, and, on average, it is 13 and 8 milliseconds for PACE and EASE, respectively. However, we found that the spawned thread finished long before the results were needed for all the benchmarks in both the modes, for both the analyses. Thus, the effective time required by the partial-results-accumulator for both PACE and EASE is zero.

Figure 16 compares the time taken by PACE with that by the existing analyzer of C2 to insert explicit null-dereference checks during JIT compilation, in milliseconds. As evident, the execution

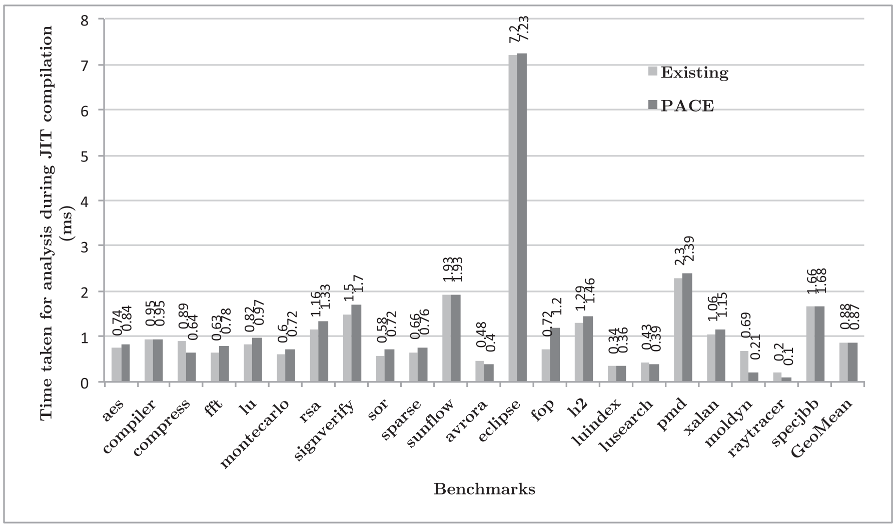


Fig. 16. Time taken (in milliseconds) to insert explicit null-dereference checks by the existing analyzer of C2 and by PACE during JIT compilation.

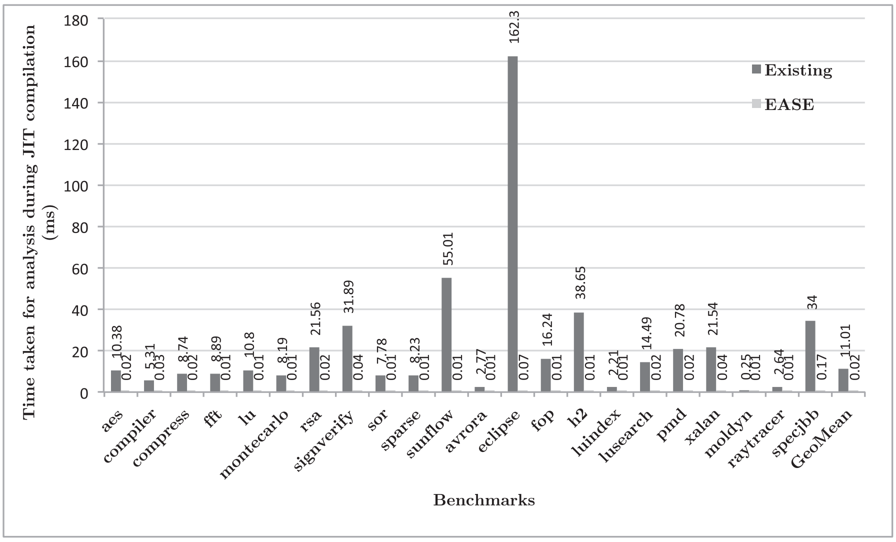


Fig. 17. Time taken (in milliseconds) to perform synchronization elimination by the existing analyzer of C2 and by EASE during JIT compilation.

times of both the analyzers are very small (less than a millisecond, on average) and comparable. The time spent by the existing analyzer is quite small because the corresponding analysis is only intraprocedural. The important point to observe is that the time spent by PACE to obtain significantly more precise results is also very small and does not cause any noticeable overhead.

Figure 17 compares the time taken by EASE with that by the existing analyzer of C2 to perform synchronization elimination during JIT compilation, in milliseconds. As evident, the escape-analysis time in EASE is clearly lower (on average, 99.91% less) compared to the existing analyzer.

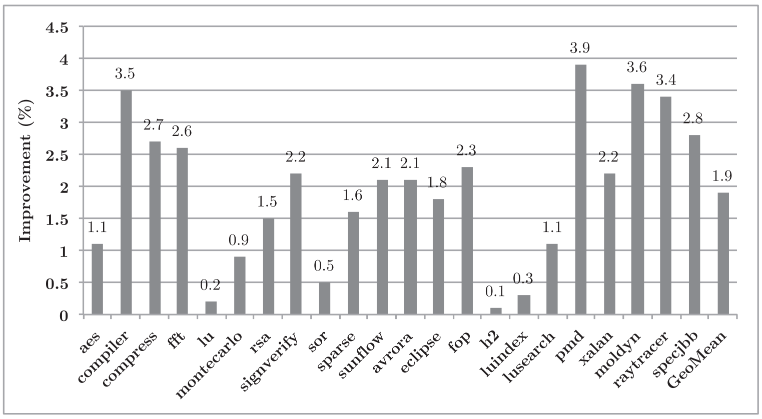


Fig. 18. Improvement in JIT compilation time for EASE.

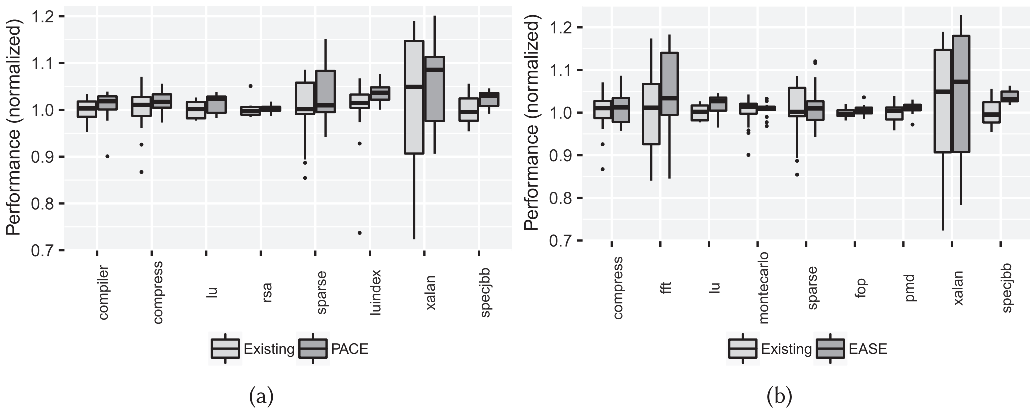


Fig. 19. Performance improvement of (a) PACE and (b) EASE, over the respective existing analyzers of C2. Benchmarks with < 0.5% improvement are not shown.

This is because EASE completely alleviates the need to construct connection graphs and propagate escape-analysis information therein, as is done by the existing analyzer of C2 during JIT compilation. Note that these savings also imply a drop in the overall JIT compilation time. For the benchmarks under consideration, we found the mean saving in the JIT compilation time due to EASE was about 1.9% (see Figure 18). Note that these improvements may also include the effects on the JIT compilation time by any additional optimizations enabled by EASE (for example, in the IR simplification passes).

7.3.3 Impact of Null-check Elimination and Synchronization Elimination. Figure 19 shows the performance improvements obtained using PACE and EASE over the respective existing analyzers of C2 in the presence of all other optimizations of the HotSpot JVM. It is well known (Georges et al. 2007) that the performance of Java programs varies significantly across multiple iterations and even across JVM invocations due to several non-deterministic factors. To measure the steady-state performance, we ran each benchmark as follows. For the SPECjvm and the DaCapo benchmarks, we executed K (=11) warm-up iterations and used the following iteration to measure the performance metric: operations-per-second for SPECjvm and time taken for DaCapo. For SPECjbb, we used the score (in operations-per-second) reported by the benchmark harness (after warming up) in each run. For the JGF benchmarks, we use the time for each run as calculated using the `time` command. Finally, in order to account for systemic bias and the variations across JVM invocations, we ran the three analyses in alternating order 20 times, and showed the variations as box plots (in Figure 19), normalized over the mean of the 20 runs of the corresponding base version (labeled "Existing" in the plots). We report only those benchmarks where the mean performance difference was more than 0.5% (thereby accounting for possible noise).

For PACE, we find that the geomean performance improvements for most benchmarks are modest: about 1%, except for SPARSE (3.5%), LUINDEX (3.4%), and SPECJBB (3.6%). Note that though the improvements for XALAN (geomean 5.7%) look comparatively high, we also observe high variance in its execution times across both versions (Existing and PACE in Figure 19(a)); this makes it difficult to conclude much about this benchmark. As most null-checks in the HotSpot JVM are actually handled using hardware traps (whose cost is not very high), the improvements obtained by PACE are along the expected lines.

For EASE (see Figure 19(b)), the mean performance improvements go up to 4.4% (for FFT), the geomean being 1.79% (for the shown benchmarks). For the benchmarks where EASE elided some synchronization operations but the improvements were negligible (e.g., SUNFLOW), we found that most elided synchronization operations were associated with static synchronized methods that were called infrequently.

Considering that the above performance improvements are observed in the presence of a host of other optimizations performed by the HotSpot JVM, we believe that these gains are important. Note that PACE and EASE may improve the performance of an application in two ways: (i) by saving time during JIT compilation (Section 7.3.2); and (ii) by enabling additional optimizations due to the enhanced precision (Section 7.3.1). The actual performance gains depend on multiple runtime factors, such as the number of times the statements containing the elided null-checks and synchronization operations are actually executed (post-compilation) at runtime, the overall execution time, and so on.

7.3.4 Comparison with Whole-program Analysis During JIT Compilation. An alternative approach to performing precise analyses in JIT compilers could be to create separate "analysis-threads" that analyze the methods being compiled in the background. However, such an approach is impractical as: (i) the time taken to perform precise analyses can be prohibitively high; and (ii) the analysis-threads may reduce the amount of parallelism available to the application. To establish this argument, we tried to perform a whole-program context-, flow- and field-sensitive analysis including the libraries statically in Soot. We set the cutoff to perform such an analysis as twice the actual execution-time of the benchmarks under consideration. Let alone the larger SPECjvm and DaCapo benchmarks, we found that the analysis for even our smallest benchmark MOLDYN did not terminate within the set cutoff. Thus, performing such expensive analyses during JIT compilation would take more time than the actual program-execution time itself and is fundamentally impractical.

Overall, we see that PYE can be used to perform highly precise program analyses without incurring any significant overheads during JIT compilation. The evaluation of PACE and EASE establishes them as practical alternatives for the existing analyzers of C2. We also note that as the overheads involved are quite small, the partial summaries generated using PYE can be used to enable sophisticated optimizations, which are currently performed only by complex JIT compilers such as C2, in faster compilers such as C1, and possibly in the interpreter as well. Even though we have implemented PYE and its instantiations in Soot and the HotSpot JVM, the proposed techniques can as well be implemented in other static analyzers such as WALA (2018), and other Java runtime environments such as the Jikes RVM (Alpern et al. 2005).

8 RELATED WORK

We divide the discussion on the related work into four parts: (i) staged analysis; (ii) modular analysis; (iii) points-to analysis for null-check elimination; and (iv) escape analysis for synchronization elimination.

8.1 Staged Analysis

There have been prior works (Ali 2014; Serrano et al. 2000) that help perform costly whole-program analyses/optimizations statically for Java. Serrano et al. (2000) propose an interesting compilation scheme in which the application (considered along with the statically available libraries) is statically compiled to a platform-specific optimized binary. This may involve dynamic compilation to support dynamic Java features (such as different runtime libraries). Averroes (Ali 2014) helps perform whole-program analyses statically by generating "placeholder" libraries that conservatively approximate the behavior of the actual runtime libraries.

Many prior works have tried to reduce the overheads at runtime (but not during JIT compilation) by taking advantage of the multi-stage nature of Java compilation/execution model. For example, Sreedhar et al. (2000) use code specialization to generate multiple versions of code statically, where each version may be optimized differently (not all may be "semantics preserving" or valid). Based on the runtime conditions, one of the valid versions of the code is invoked during execution. Similarly, Guyer et al. (2006) annotate the input code with explicit "free" instructions (executed at runtime) based on the liveness information of the heap objects. In contrast, we perform expensive analyses on applications statically to obtain results that are conditional on the specific libraries on the target machine; these partial results are combined with the partial results of the libraries, at runtime, to achieve precision and performance.

Chambers (2002) and Philipose et al. (2002) propose a staged compilation scheme in which the generation of native code for different platforms is spread across the different stages of compilation and linking, for C programs. In contrast, PYE is designed for analyzing programs written in languages like Java/C# that follow a two-step process of compilation (static + just-in-time), where the libraries can only be obtained at runtime.

Sharma et al. (1998) propose *deferred* data-flow analysis (DDFA) to address the problem of the conservative nature of static analyses. DDFA performs most of the analysis at compile-time and uses control-flow information at runtime to improve the precision of the analysis. In PYE, our focus is on handling the dependence between the application and the library methods. It would be interesting to perform DDFA analyses in the PYE framework.

Our idea of conditional values is partially inspired from the idea of three-valued logic analysis (Sagiv et al. 2002). We associate conditions with the indeterminate third values, which are resolved during JIT compilation. This helps us achieve precision without much overhead at runtime.

8.2 Modular Analysis

Modular analysis, as proposed by Cousot and Cousot (2002), is a well-explored technique to scale interprocedural analyses by analyzing different modules (or methods, in Java context) separately, and composing the modular results to obtain whole-program analysis results (Calcagno et al. 2011; Choi et al. 1999; Sălcianu and Rinard 2005; Vivien and Rinard 2001; Whaley and Rinard 1999). There is a recent survey by Madhavan et al. (2015) that evaluates several of these existing techniques in a well-formalized framework.

In the context of Java programs, Whaley and Rinard (1999) compute summaries for methods in the form of points-to escape graphs, and compose them at interprocedural boundaries. Later, Vivien and Rinard (2001) incrementalize the analysis to analyze only those parts of a program that may deliver useful results. The analysis performed by our proposed framework PYE is also modular, but we do not have the library methods while analyzing the application, and vice-versa. Consequently, our generated summaries contain conditional values. In PACE and EASE, we have borrowed the idea of creating outside nodes from Whaley and Rinard (1999) and added conditional values to the outside nodes to represent the dependence of unavailable object-dereferences on the unanalyzed parts of a program. Further, we use the mapping algorithm presented by Whaley and Rinard (1999) to merge the points-to graphs of analyzed methods at interprocedural boundaries.

There have been prior works that model the dependences between the available and unavailable methods while performing static analyses. WALA (2018) models native methods flow-insensitively and merges their summaries with those of their callers. StubDroid (Arzt and Bodden 2016) summarizes Android libraries for taint analysis by storing conditions on the "taint value" of actual arguments. In contrast, the conditional values proposed in PYE are bidirectional (that is, from application to library methods and vice-versa), and the statically generated partial summaries are resolved during JIT compilation to obtain precise analysis-results in the JVM.

8.3 Points-to Analysis for Null-check Elimination

There have been works that perform points-to analysis to statically identify unnecessary null-dereference checks (Loginov et al. 2008; Nanda and Sinha 2009). Loginov et al. (2008) performe a static points-to analysis and annotate the statements that are guaranteed to dereference a concrete object. They handle those library calls precisely whose specifications guarantee that the corresponding methods return a non-null object, and treat others conservatively. Nanda and Sinha (2009) perform a path-sensitive points-to analysis statically to mark the dereferences guaranteed to be made on a concrete object. They treat the library methods as unavailable for analysis and handle the library calls conservatively. Contrary to both these works, PACE is a two-step analysis that neither assumes the specification of library methods nor handles library calls conservatively. Instead, PACE analyzes the application and the library methods separately, while encoding the dependence between them in the generated partial summaries as conditional values. These dependences are resolved during JIT compilation to obtain precise analysis-results.

In order to balance the time spent in JIT compilation, the HotSpot Server Compiler (C2) of the HotSpot JVM (Paleczny et al. 2001) performs an intraprocedural points-to analysis to avoid inserting null-checks that are not required. As shown in Section 7, PACE offers a much enhanced precision (on average 23.67% fewer checks than the existing analyzer) in almost the same amount of time (as C2) during JIT compilation.

8.4 Escape Analysis for Synchronization Elimination

There have been many prior research works (Choi et al. 1999; Salcianu and Rinard 2001; Whaley and Rinard 1999) that perform precise escape analysis for Java programs. However, they are completely performed either during static compilation (Choi et al. 1999) (make conservative assumptions about libraries), or during JIT compilation (Salcianu and Rinard 2001; Whaley and Rinard 1999) (not scalable). Kotzmann and Mössenböck (2005) present an imprecise but fast escape analysis for the HotSpot Client Compiler (C1). In contrast, EASE generates precise escape-analysis results during JIT compilation in the HotSpot Server Compiler, at speeds comparable to that of the baseline partially interprocedural and partially flow-sensitive analysis.

Lee and Midkiff (2006) propose an insightful two-phase escape analysis for the Jikes RVM. They compute connection graphs (a representation similar to points-to graphs) for different methods offline and merge the connection graphs to complete an interprocedural analysis during JIT compilation. On the contrary, EASE generates precise escape-analysis results at runtime, by resolving the statically generated partial summaries, and has the following advantages: (i) By maintaining the set `SYN` separately, EASE can preserve flow-sensitivity for synchronization elimination (Lee and Midkiff store only per-method connection-graphs and lose flow-sensitivity). (ii) The overhead of the fast-precise-analyzer of EASE at runtime is very less, as it does not perform any actual escape analysis (Lee and Midkiff may have to revisit/modify the connection-graph multiple times). (iii) The storage overheads of the result files for EASE are quite less: average 3.96% over the class files (Lee and Midkiff report 68% overhead).

The recent proposal of *partial* escape analysis (Stadler et al. 2014) performs lock elision only on those branches in which the associated object does not escape, in the Graal (2018) compiler. For instance, before entering a synchronization statement, a data-structure is looked up to check if the associated object has not escaped; and if so, the lock operation is not performed. This is a promising approach for doing escape analysis, and we believe that its efficiency can be further improved by implementing it in PYE.

9 CONCLUSION AND FUTURE WORK

In this article, we propose a two-step analysis framework called PYE that helps generate highly precise analysis-results during JIT compilation, at a low cost. PYE is based on the idea of generating partial summaries at compile-time, which encode the dependence on the missing libraries in a concise manner, in the form of conditional values. We show the effectiveness of PYE by using it to design two precise analyses—PACE and EASE—for Java programs. Over a wide range of benchmarks, PACE and EASE generate more precise results compared to the existing analyzers of the HotSpot Server Compiler (C2), with negligible overhead (in fact, saving time in case of EASE) during JIT compilation. The evaluation attests PYE to be an effective and practical framework for implementing complicated whole-program analyses and their related optimizations. The techniques proposed in this article are general enough to be extended to other languages such as C# that deploy a two-step compilation process.

Future Work

It would be quite interesting to implement precise versions of other popular analyses (such as points-to analysis for call-graph construction, may-happen-in-parallel analysis, and so on) in PYE, and study the impact on the precision of the results obtained, possibly for other static+JIT-compiled languages (such as C#) as well.

It would also be interesting to make the partial-analyzer incremental, such that the partial summaries could be updated as the program changes, on-the-fly, without performing the static analysis from scratch. Such an extension can be useful in incremental compilation frameworks such as the Eclipse Java Compiler (ECJ) used along with the Eclipse IDE (Eclipse 2018).

ACKNOWLEDGMENTS

We would like to thank the ACM TOPLAS associate editor Steve Blackburn and the anonymous reviewers for their comprehensive suggestions toward improving the article. We express deep gratefulness to our PACE Lab (IIT Madras) colleagues Raghesh A., Anchu R. S., Jyothi Vedurada, Aman Nougrahiya, and Rupesh Nasre for their thoughtful feedback and motivation throughout the duration of this work.

REFERENCES

Karim Ali. 2014. *The Separate Compilation Assumption*. Ph.D. Dissertation. University of Waterloo, Waterloo, Ontario, Canada. Advisor(s) Lhoták, Ondřej.

B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. 2005. The Jikes research virtual machine project: Building an open-source research community. *IBM Syst. J.* 44, 2 (Jan. 2005), 399–417.

Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language*. Technical Report.

Steven Arzt and Eric Bodden. 2016. StubDroid: Automatic inference of precise data-flow summaries for the android framework. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 725–735. DOI:https://doi.org/10.1145/2884781.2884816

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA'06)*. ACM, New York, NY, 169–190.

Bruno Blanchet. 2003. Escape analysis for JavaTM: Theory and practice. *ACM Trans. Program. Lang. Syst.* 25, 6 (Nov. 2003), 713–775. DOI:https://doi.org/10.1145/945885.945886

Eric Bodden, Andreas Sewe, Jan Sinschek, Mira Mezini, and Hela Oueslati. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceeding of the 33rd International Conference on Software Engineering (ICSE'11)*. ACM, New York, NY, 241–250. DOI:https://doi.org/10.1145/1985793.1985827

Jeff Bogda and Urs Hölzle. 1999. Removing unnecessary synchronization in Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*. ACM, New York, NY, 35–46. DOI:https://doi.org/10.1145/320384.320388

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional shape analysis by means of bi-abduction. *J. ACM* 58, 6, Article 26 (Dec. 2011), 66 pages. DOI:https://doi.org/10.1145/2049697.2049700

Craig Chambers. 2002. Staged compilation. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM'02)*. ACM, New York, NY, 1–8. DOI:https://doi.org/10.1145/503032.503045

Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. 1999. Escape analysis for Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*. ACM, New York, NY, 1–19. DOI:https://doi.org/10.1145/320384.320386

Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. 2002. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*. ACM, New York, NY, 258–269. DOI:https://doi.org/10.1145/512529.512560

Keith D. Cooper, Ken Kennedy, and Linda Torczon. 1986. Interprocedural optimization: Eliminating unnecessary recompilation. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction (SIGPLAN'86)*. ACM, New York, NY, 58–67. DOI:https://doi.org/10.1145/12276.13317

Patrick Cousot and Radhia Cousot. 2002. Modular static program analysis. In *Proceedings of the 11th International Conference on Compiler Construction (CC'02)*. Springer-Verlag, London, UK, 159–178. http://dl.acm.org/citation.cfm?id=647478.727794.

Charles Daly, Jane Horgan, James Power, and John Waldron. 2001. Platform independent dynamic Java virtual machine analysis: The Java grande forum benchmark suite. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande (JGI'01)*. ACM, New York, NY, 106–115. DOI:https://doi.org/10.1145/376656.376826

Jens Dietrich, Nicholas Hollingum, and Bernhard Scholz. 2015. Giga-scale exhaustive points-to analysis for Java in under a minute. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'15)*. ACM, New York, NY, 535–551. DOI:https://doi.org/10.1145/2814270.2814307

Tamar Domani, Gal Goldshtein, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. 2002. Thread-local heaps for Java. In *Proceedings of the 3rd International Symposium on Memory Management (ISMM'02)*. ACM, New York, NY, 76–87. DOI:https://doi.org/10.1145/512429.512439

Eclipse. 2018. Eclipse IDE. Retrieved from http://www.eclipse.org/.

Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous Java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA'07)*. ACM, New York, NY, 57–76. DOI:https://doi.org/10.1145/1297027.1297033

Graal. 2018. OpenJDK Graal. Retrieved from http://openjdk.java.net/projects/graal/.

Samuel Z. Guyer, Kathryn S. McKinley, and Daniel Frampton. 2006. Free-me: A static analysis for automatic individual object reclamation. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06).* ACM, New York, NY, 364–375. DOI:https://doi.org/10.1145/1133981.1134024

Ben Hardekopf and Calvin Lin. 2007. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07).* ACM, New York, NY, 290–299. DOI:https://doi.org/10.1145/1250734.1250767

Uday P. Khedker and Bageshri Karkare. 2008. Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: Resurrecting the classical call strings method. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction (CC'08/ETAPS'08).* Springer-Verlag, Berlin, 213–228. http://dl.acm.org/citation.cfm?id=1788374.1788394.

Thomas Kotzmann and Hanspeter Mössenböck. 2005. Escape analysis in the context of dynamic compilation and deoptimization. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE'05).* ACM, New York, NY, 111–120. DOI:https://doi.org/10.1145/1064979.1064996

Kyungwoo Lee and Samuel P. Midkiff. 2006. A two-phase escape analysis for parallel Java programs. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT'06).* ACM, New York, NY, 53–62. DOI:https://doi.org/10.1145/1152154.1152166

Alexey Loginov, Eran Yahav, Satish Chandra, Stephen Fink, Noam Rinetzky, and Mangala Nanda. 2008. Verifying dereference safety via expanding-scope analysis. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA'08).* ACM, New York, NY, 213–224. DOI:https://doi.org/10.1145/1390630.1390657

Ravichandhran Madhavan, G. Ramalingam, and Kapil Vaswani. 2015. A framework for efficient modular heap analysis. *Found. Trends Program. Lang.* 1, 4 (Jan. 2015), 269–381. DOI:https://doi.org/10.1561/2500000020

Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation.* Morgan Kaufmann.

Mangala Gowri Nanda and Saurabh Sinha. 2009. Accurate interprocedural null-dereference analysis for Java. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09).* IEEE Computer Society, Washington, DC, 133–143. DOI:https://doi.org/10.1109/ICSE.2009.5070515

Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. 1999. An efficient algorithm for computing MHP information for concurrent Java programs. In *Proceedings of the 7th European Software Engineering Conference/7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-7).* Springer-Verlag, London, UK, 338–354. http://dl.acm.org/citation.cfm?id=318773.319252.

Erik M. Nystrom, Hong-Seok Kim, and Wen-mei W. Hwu. 2004. Importance of heap specialization in pointer analysis. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'04).* ACM, New York, NY, 43–48. DOI:https://doi.org/10.1145/996821.996836

Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java HotSpot^{TM} server compiler. In *Proceedings of the 2001 Symposium on Java^{TM} Virtual Machine Research and Technology Symposium—Volume 1 (JVM'01).* USENIX Association, Berkeley, CA, 1–1. http://dl.acm.org/citation.cfm?id=1267847.1267848.

Matthai Philipose, Craig Chambers, and Susan J. Eggers. 2002. Towards automatic construction of staged compilers. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02).* ACM, New York, NY, 113–125. DOI:https://doi.org/10.1145/503272.503284

Erik Ruf. 2000. Effective synchronization removal for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI'00).* ACM, New York, NY, 208–218. DOI:https://doi.org/10.1145/349299.349327

Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. 2002. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.* 24, 3 (May 2002), 217–298.

Alexandru Salcianu and Martin Rinard. 2001. Pointer and escape analysis for multithreaded programs. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP'01).* ACM, New York, NY, 12–23. DOI:https://doi.org/10.1145/379539.379553

Mauricio Serrano, Rajesh Bordawekar, Sam Midkiff, and Manish Gupta. 2000. Quicksilver: A quasi-static compiler for Java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'00).* ACM, New York, NY, 66–82. DOI:https://doi.org/10.1145/353171.353176

M. Sharir and A. Pnueli. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications, ch. 7.* Prentice- Hall.

Shamik D. Sharma, Anurag Acharya, and Joel Saltz. 1998. *Deferred Data-Flow Analysis.* Technical Report. University of Maryland.

SPECjbb. 2005. SPEC JBB2005. Retrieved from https://www.spec.org/jbb2005/.

SPECjvm. 2008. SPEC JVM2008. Retrieved from https://www.spec.org/jvm2008/.

Vugranam C. Sreedhar, Michael Burke, and Jong-Deok Choi. 2000. A framework for interprocedural optimization in the presence of dynamic class loading. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI'00).* ACM, New York, NY, 196–207. DOI:https://doi.org/10.1145/349299.349326

Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial escape analysis and scalar replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'14)*. ACM, New York, NY, Article 165, 10 pages. DOI:https://doi.org/10.1145/2544137.2544157

Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*. ACM, New York, NY, 32–41. DOI:https://doi.org/10.1145/237721.237727

Douglas R. Stinson. 1995. *Cryptography: Theory and Practice* (1st ed.). CRC Press, Inc., Boca Raton, FL.

Alexandru Sălcianu and Martin Rinard. 2005. Purity and side effect analysis for Java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*. Springer-Verlag, Berlin, 199–215. DOI:https://doi.org/10.1007/978-3-540-30579-8_14

Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and precise points-to analysis: Modeling the heap by merging equivalent automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17)*. ACM, New York, NY, 278–291. DOI:https://doi.org/10.1145/3062341.3062360

Manas Thakur and V. Krishna Nandivada. 2019. Compare less, defer more: Scaling value-contexts based whole-program heap analyses. In *Proceedings of the 28th International Conference on Compiler Construction (CC'19)*. ACM, New York, NY, 135–146. DOI:https://doi.org/10.1145/3302516.3307359

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot—A Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'99)*. IBM Press, 13–. http://dl.acm.org/citation.cfm?id=781995.782008.

Frédéric Vivien and Martin Rinard. 2001. Incrementalized pointer and escape analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI'01)*. ACM, New York, NY, 35–46. DOI:https://doi.org/10.1145/378795.378804

WALA. 2018. The T. J. Watson Libraries for Analysis. Retrieved from http://wala.sourceforge.net.

John Whaley and Martin Rinard. 1999. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*. ACM, New York, NY, 187–206.

Received April 2018; revised February 2019; accepted April 2019